CORE JAVA

**Serialization** is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

# Static Blocks:

Static block is mostly used for changing the default values of static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

# Difference between Set, List and Map in Java - Interview question

Set, List and Map are three important interface of Java collection framework and Difference between Set, List and Map in Java is one of the most frequently asked [Java Collection interview question](http://java67.blogspot.com/2012/09/java-collection-interview-questions.html). Sometime this question is asked as When to use List, Set and Map in Java. Clearly, interviewer is looking to know that whether you are familiar with fundamentals of Java collection framework or not. In order to decide when to use List, Set or Map, you need to know what are these interfaces and what functionality they provide. [List in Java](http://java67.blogspot.com/2012/07/sort-list-ascending-descending-order-set-arraylist.html) provides ordered and indexed collection which may contain duplicates. Set provides an un-ordered collection of unique objects, i.e. Set doesn't allow duplicates, while Map provides a data structure based on key value pair and hashing. All three List, Set and Map are interfaces in Java and there are many concrete implementation of them are available in Collection API. ArrayList and LinkedList are two most popular used List implementation while [LinkedHashSet, TreeSet and HashSet](http://javarevisited.blogspot.com/2012/11/difference-between-treeset-hashset-vs-linkedhashset-java.html) are frequently used Set implementation. In this Java article we will see difference between Map, Set and List in Java and learn when to use List, Set or Map.

## Set vs List vs Map in Java

As I said Set, List and Map are interfaces, which defines core contract e.g. a Set contract says that it cannot contain duplicates. Based upon our knowledge of List, Set and Map let's compare them on different metrics.

**Duplicate Objects**

Main difference between List and Set interface in Java is that List allows duplicates while Set doesn't allow duplicates. All implementation of Set honor this contract. Map holds two object per Entry e.g. key and value and It may contain duplicate values but keys are always unique. See [here](http://java67.blogspot.sg/2012/08/difference-between-list-and-set-in-java.html) for more difference between List and Set data structure in Java.

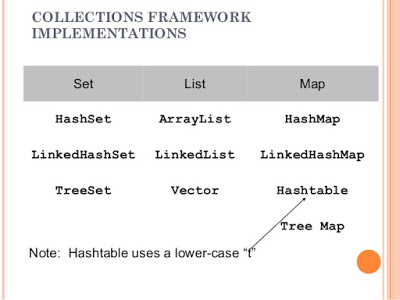
**Order**

Another key difference between List and Set is that List is an ordered collection, List's contract maintains insertion order or element. Set is an unordered collection, you get no guarantee on which order element will be stored. Though some of the Set implementation e.g. LinkedHashSet maintains order. Also SortedSet and SortedMap e.g. [TreeSet and TreeMap](http://java67.blogspot.com/2012/08/difference-between-treemap-and-treeset-java.html) maintains a sorting order, imposed by using Comparator or Comparable.

**Null elements**

List allows null elements and you can have many null objects in a List, because it also allowed duplicates. Set just allow one null element as there is no duplicate permitted while in Map you can have null values and at most one null key. worth noting is that [Hashtable doesn't allow null key or values](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) but HashMap allows null values and one null keys.  This is also the main difference between these two popular implementation of Map interface, aka HashMap vs Hashtable.

**Popular implementation**  
Most popular implementations of List interface in Java are ArrayList, LinkedList and Vector class. ArrayList is more general purpose and provides random access with index, while LinkedList is more suitable for frequently adding and removing elements from List. Vector is synchronized counterpart of ArrayList. On the other hand, most popular implementations of Set interface are HashSet, LinkedHashSet and TreeSet. First one is general purpose Set which is backed by HashMap , see [how HashSet works internally in Java](http://java67.blogspot.sg/2014/01/how-hashset-is-implemented-or-works-internally-java.html) for more details. It also doesn't provide any ordering guarantee but LinkedHashSet does provides ordering along with uniqueness offered by Set interface. Third implementation TreeSet is also an implementation of SortedSet interface, hence it keep elements in a sorted order specified by compare() or compareTo() method. Now the last one, most popular implementation of Map interface are HashMap, LinkedHashMap, Hashtable and TreeMap.  First one is the non synchronized general purpose Map implementation while Hashtable is its synchronized counterpart, both doesn't provide any ordering guarantee which comes from LinkedHashMap. Just like TreeSet, TreeMap is also a sorted data structure and keeps keys in sorted order.

[](http://1.bp.blogspot.com/-uarJzYtW7Ho/VXRgq-suh5I/AAAAAAAAC8U/GB8MqbvYgpU/s1600/Difference+between+Set%2C+List+and+Map+in+Java.jpg)

### When to use List, Set and Map in Java

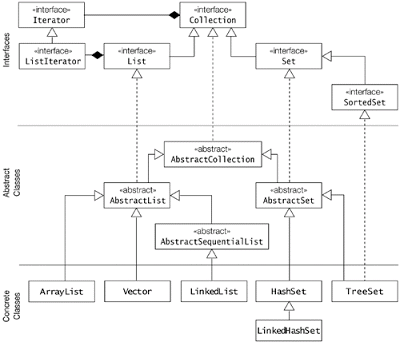
Based upon our understanding of difference between Set, List and Map we can now decide when to use List, Set or Map in Java.

1) If you need to access elements frequently by using index, than List is a way to go. Its implementation e.g. [ArrayList](http://javarevisited.blogspot.com/2011/05/example-of-arraylist-in-java-tutorial.html) provides faster access if you know index.

2) If you want to store elements and want them to maintain an order on which they are inserted into collection then go for List again, as List is an ordered collection and maintain insertion order.

3) If you want to create collection of unique elements and don't want any duplicate than choose any Set implementation e.g. HashSet, LinkedHashSet or TreeSet. All Set implementation follow there general contract e.g. uniqueness but also add addition feature e.g. TreeSet is a SortedSet and elements stored on TreeSet can be sorted by using [Comparator or Comparable in Java](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html). LinkedHashSet also maintains insertion order.

4) If you store data in form of key and value than Map is the way to go. You can choose from Hashtable, HashMap, TreeMap based upon your subsequent need. In order to choose between first two see [difference between HashSet and HashMap in Java](http://javarevisited.blogspot.com/2011/09/difference-hashmap-vs-hashset-java.html).

[](http://1.bp.blogspot.com/-_DivoplP9to/VXRlC2h2_eI/AAAAAAAAC8s/jWyoFfnjz-M/s1600/Java+Collection+framework+Map+Set+and+List.gif)

That's all on difference between Set, List and Map in Java. All three are most fundamental interface of Java Collection framework and any Java developer should know there distinguish feature and given a situation should be able to pick right Collection class to use. It's also good to remember difference between there implementation e.g. [When to use ArrayList and LinkedList](http://java67.blogspot.com/2012/12/difference-between-arraylist-vs-LinkedList-java.html) , [HashMap vs Hashtable](http://javarevisited.blogspot.sg/2010/10/difference-between-hashmap-and.html) or [When to use Vector or ArrayList](http://java67.blogspot.com/2012/09/arraylist-vs-vector-in-java-interview.html) etc. Collection API is huge and it's difficult to know every bits and piece but at same time there is no excuse for not knowing fundamentals like difference between Set, List and Map in Java.

List and Set both are interfaces. They both extends Collection interface. In this post we are discussing the **differences between List and Set interfaces** in java.

## List Vs Set

1) List is an ordered collection it maintains the insertion order, which means upon displaying the list content it will display the elements in the same order in which they got inserted into the list.

Set is an unordered collection, it doesn’t maintain any order. There are few implementations of Set which maintains the order such as LinkedHashSet (It maintains the elements in insertion order).

2) List allows duplicates while Set doesn’t allow duplicate elements. All the elements of a Set should be unique if you try to insert the duplicate element in Set it would replace the existing value.

3) List implementations: [ArrayList](http://beginnersbook.com/2013/12/java-arraylist/), [LinkedList](http://beginnersbook.com/2013/12/linkedlist-in-java-with-example/) etc.

4. Set implementations: [HashSet](http://beginnersbook.com/2013/12/hashset-class-in-java-with-example/), [LinkedHashSet](http://beginnersbook.com/2013/12/linkedhashset-class-in-java-with-example/), [TreeSet](http://beginnersbook.com/2013/12/treeset-class-in-java-with-example/) etc.

4) List allows any number of null values. Set can have only a single null value at most.

5) [ListIterator](http://beginnersbook.com/2014/06/listiterator-in-java-with-examples/) can be used to traverse a List in both the directions(forward and backward) however it cannot be used to traverse a Set. We can use [Iterator](http://beginnersbook.com/2014/06/java-iterator-with-examples/) (It works with List too) to traverse a Set.

6) List interface has one legacy class called [Vector](http://beginnersbook.com/2013/12/vector-in-java/)whereas Set interface does not have any legacy class.

## When to use Set and When to use List?

The usage is purely depends on the requirement:

If the requirement is to have only unique values then Set is your best bet as any implementation of Set maintains unique values only.

If there is a need to maintain the insertion order irrespective of the duplicity then List is a best option. Both the implementations of List interface – ArrayList and LinkedList sorts the elements in their insertion order.

**What is static import?**

If we have to use any static variable or method from other class, usually we import the class and then use the method/variable with class name.

import java.lang.Math;

//inside class

double test = Math.PI \* 5;

We can do the same thing by importing the static method or variable only and then use it in the class as if it belongs to it.

import static java.lang.Math.PI;

//no need to refer class now

double test = PI \* 5;

Use of static import can cause confusion, so it’s better to avoid it. Overuse of static import can make your program unreadable and unmaintainable.

**What is the benefit of Composition over Inheritance?**

One of the best practices of java programming is to “favor composition over inheritance”. Some of the possible reasons are:

* Any change in the superclass might affect subclass even though we might not be using the superclass methods. For example, if we have a method test() in subclass and suddenly somebody introduces a method test() in superclass, we will get compilation errors in subclass. Composition will never face this issue because we are using only what methods we need.
* Inheritance exposes all the super class methods and variables to client and if we have no control in designing superclass, it can lead to security holes. Composition allows us to provide restricted access to the methods and hence more secure.
* We can get runtime binding in composition where inheritance binds the classes at compile time. So composition provides flexibility in invocation of methods.

**What is Classloader in Java?**

Java Classloader is the program that loads byte code program into memory when we want to access any class. We can create our own class loader by extending ClassLoader class and overriding loadClass(String name) method.

**What are different types of classloaders?**

There are three types of built-in Class Loaders in Java:

Bootstrap Class Loader – It loads JDK internal classes, typically loads rt.jar and other core classes.

Extensions Class Loader – It loads classes from the JDK extensions directory, usually $JAVA\_HOME/lib/ext directory.

System Class Loader – It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options.

## What is Marker interfaces in Java and why required?

### Why Marker or Tag interface do in Java?

Looking carefully on marker interface in Java e.g. [**Serializable**](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html), **Cloneable** and **Remote** it looks they are used to indicate something to compiler or JVM. So if JVM sees a Class is Serializable it done some special operation on it, similar way if JVM sees one Class is implement Cloneable it performs some operation to support cloning. Same is true for RMI and Remote interface. So in short Marker interface indicate, signal or a command to Compiler or [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html).

This is pretty standard answer of question about marker interface and once you give this answer most of the time interviewee definitely asked "Why this indication cannot be done using a flag inside a class?” this make sense right? Yes this can be done by using a boolean flag or a String but doesn't marking a class like **Serializable** or **Cloneable** makes it more readable and it also allows to take advantage [of Polymorphism in Java](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html).

Where Should I use Marker interface in Java

Apart from using built in marker interface for making a class Serializable or Cloneable. One can also develop his own marker interface. Marker interface is a good way to classify code. You can create marker interface to logically divide your code and if you have your own tool than you can perform some pre-processing operation on those classes. Particularly useful for developing API and framework like [Spring](http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html) or [Struts](http://javarevisited.blogspot.com/2011/11/struts-interview-questions-answer-j2ee.html).

After introduction of Annotation on Java5, Annotation is better choice than marker interface and **JUnit** is a perfect example of using Annotation e.g. @Test for specifying a Test Class. Same can also be achieved by using Test marker interface.

**Another use of marker interface in Java**

One more use of marker interface in Java can be commenting. a marker interface called ThreadSafe can be used to communicate other developers that classes implementing this marker interface gives thread-safe guarantee and any modification should not violate that. Marker interface can also help code coverage or code review tool to find bugs based on specified behavior of marker interfaces.

Again Annotations are better choice @ThreadSafe looks lot better than implementing **ThraedSafe** marker interface.

In summary marker interface in Java is used to indicate something to compiler, JVM or any other tool but Annotation is better way of doing same thing.

What is the difference between the JRE and the JDK?

|  |  |
| --- | --- |
| JRE  (Java Runtime environment) | JDK (Java Development Kit) |
| It is an implementation of the Java Virtual Machine\* which actually executes Java programs. | It is a bundle of software that you can use to develop Java based applications. |
| Java Runtime Environment is a plug-in needed for running java programs. | Java Development Kit is needed for developing java applications. |
| The JRE is smaller than the JDK so it needs less Disk space. | The JDK needs more Disk space as it contains the JRE along with various development tools. |
| The JRE can be downloaded/supported freely from [java.com](http://www.java.com/) | The JDK can be downloaded/supported freely from [oracle.com/technetwork/java/javase/downloads/](http://www.oracle.com/technetwork/java/javase/downloads/index.html) |
| It includes the JVM, Core libraries and other additional components to run applications and applets written in Java. | It includes the JRE, set of API classes, Java compiler, Webstart and additional files needed to write Java applets and applications. |

What is the difference between the JRE and the Java SE platform?

|  |  |  |
| --- | --- | --- |
|  | JRE  (Java Runtime Environment) | Java SE (Java Platform, Standard Edition) |
| Who needs it? | Computer users who run applets and applications written using Java technology | Software developers who write applets and applications using Java technology |
| What is it? | An environment required to run applets and applications written using the Java programming language | A software development kit used to write applets and applications using the Java programming language |
| How do you get it? | Distributed freely and is available from:  [java.com](https://www.java.com/en/) | Distributed freely and is available from:  [oracle.com/javase](http://oracle.com/javase) |

# [AtomicInteger in Java](http://www.instanceofjava.com/2015/12/atomic-integer-example-java-interview.html)

* Java.util.concurrent.atomic package provides very useful classes that support lock free and thread safe programming.
* The main use of this class is an int value that may be updated automatically.
* AtomicInteger has some useful methods. Before that let’s see the some points about this class.
* Commonly we will use this AtomicInteger to handle the counter that is accessible by different threads simultaneously.

**Why You Need String Constant Pool? :**

* String objects in java are stored in two places in memory. One is String Constant Pool and another one is Heap Memory. String objects created using string literals are stored in String Constant Pool where as string objects created using new operator are stored in heap memory.
* String objects are most used objects in the development of any kind of applications. Therefore, there has to be a special arrangement to store these objects. String Constant Pool is one such special arrangement. In string constant pool, there will be no two objects with the same content. Heap memory can have any number of objects with same content.
* Just imagine creating 1000 string objects with same content in heap memory and one string object with that content in String Constant Pool. Which one saves the memory? Which one will save the time?. Which one will be accessed faster?. It is, of course, String Constant Pool. That’s why you need String Constant Pool.

**What Is String Intern? :**

String intern or simply intern refers to string object in the String Constant Pool. Interning is the process of creating a string object in String Constant Pool which will be exact copy of string object in heap memory.

**What is the use of interning the string?**

**To save the memory Space:**

Using interned string, you can save the memory space. If you are using lots of string objects with same content in your code, than it is better to create an intern of that string in the pool. Use that intern string whenever you need it instead of creating a new object in the heap. It saves the memory space.

**For Faster Comparison :**

Assume that there are two string objects s1 and s2 in heap memory and you need to perform comparison of these two objects more often in your code. Then using s1.intern() == s2.intern() will be more fast then s1.equals(s2). Because, equals() method performs character by character comparison where as “==” operator just compares references of objects.

**STRING:**

String represents sequence of characters enclosed within the double quotes.  “abc”, “JAVA”, “123”, “A” are some examples of strings. In many languages, strings are treated as character arrays. But In java, strings are treated as objects. To create and manipulate the strings, Java provides three classes.

**1) java.lang.String                  (From JDK 1.0)**

**2) java.lang.StringBuffer            (From JDK 1.5)**

**3) java.lang.StringBuilder           (From JDK 1.5)**

1) All these three classes are members of java.lang package and they are final classes. That means you can’t create subclasses to these three classes.

2) All three classes implement Serializable and CharSequence interface.

3) java.lang.String objects are immutable in java. That is, once you create String objects, you can’t modify them. Whenever you try to modify the existing String object, a new String object is created with modifications. Existing object is not at all altered. Where as java.lang.StringBuffer and java.lang.StringBuilder objects are mutable. That means, you can perform modifications to existing objects.

4) Only String and StringBuffer objects are thread safe. StringBuilder objects are not thread safe. So whenever you want immutable and thread safe string objects, use java.lang.String class and whenever you want mutable as well as thread safe string objects then use java.lang.StringBuffer class.

5) In all three classes, toString() method is overrided. So, whenever you use reference variables of these three types, they will return contents of the objects not physical address of the objects.

## 6) hashCode() and equals() methods are overrided only in java.lang.String class but not in java.lang.StringBuffer and java.lang.StringBuilder classes.

7) There is no reverse() and delete() methods in String class. But, String

Buffer and StringBuilder have reverse() and delete() methods.

8) In case of String class, you can create the objects without new operator. But in case of StringBuffer and StringBuilder class, you have to use new operator to create the objects.

**Are string objects created using new operator also immutable?**

The answer is yes. String objects created using new operator are also immutable although they are stored in the heap memory. This can be also proved with help of an example.

|  |  |
| --- | --- |
|  | public class StringExamples  {      public static void main(String[] args)      {          String s1 = new String("JAVA");            System.out.println(s1);         //Output : JAVA            s1.concat("J2EE");            System.out.println(s1);         //Output : JAVA      }  } |

In this example, a string object is created with “JAVA” as its content using new operator and its reference is assigned to s1. I have tried to change the contents of this object using concat() method. But, these changes are not reflected in the object as seen in Line 11. Even after the concatenation, content of the object is same as before. This is because the strings are immutable. Once I tried to concatenate “J2EE” to an existing string “JAVA”, a new string object is created with “JAVAJ2EE” as it’s content. But we don’t have reference to that object in this program.

Conclusion:

Immutability is the fundamental property of string objects. In whatever way you create the string objects, either using string literals or using new operator, they are immutable.

## Process :

Process is an executing instance of an application. For example, when you double click MS Word icon in your computer, you start a process that will run this MS word application. Processes are heavy weight operations that they require their own separate memory address in operating system. Because of the processes are stored in separate memory, communication between processes (Inter Process Communication) takes time. Context switching from one process to another process is also expensive.

## Thread :

Thread is a smallest executable unit of a process. Thread has it’s own path of execution in a process. For example, when you start MS word, operating system creates a process and start the execution of a primary thread of that process. A process can have multiple threads. Threads of the same process share the memory address of that process. i.e threads are stored inside the memory of a process. As the threads are stored in the same memory space, communication between threads (Inter Thread Communication) is fast. Context switching from one thread to another thread is also less expensive.

**There are two types of Threads in java**.

1**) User Thread**

**2) Daemon Thread**

**1) User Thread:**

User threads are threads which are created by the application or user. They are high priority threads. JVM (Java Virtual Machine) will not exist until all user threads finish their execution. JVM wait for these threads to finish their task. These threads are foreground threads.

2) **Daemon Thread:**

Daemon threads are threads which are mostly created by the JVM. These threads always run in background. These threads are used to perform some background tasks like garbage collection and house-keeping tasks. These threads are less priority threads. JVM will not wait for these threads to finish their execution. JVM will exit as soon as all user threads finish their execution. JVM doesn’t wait for daemon threads to finish their task.

[**How to identify a thread in java**?](http://javaconceptoftheday.com/how-to-identify-a-thread-in-java/)

In a multithreaded application, It is very important to know which thread is currently executing it’s task. But the question is, How to identify a thread?. The answer which effortlessly comes to our mind is “through it’s name”. Of course, you can identify a thread by it’s name.  But, more than one threads can have the same name. This makes identifying a thread more difficult. There is a solution for this problem from JDK 1.5 onward. JVM assigns one unique long number for every thread created. This remains unchanged for the whole life term of a thread. This number can be used to identify a thread.

From JDK 1.5 onward, One more method added to java.lang.Thread class. That is getID () method. This method returns the unique long number associated with a thread. That can be used as an identifier of a thread. Below is the method signature of getID () method.

**What is Memory Leak in Java?**

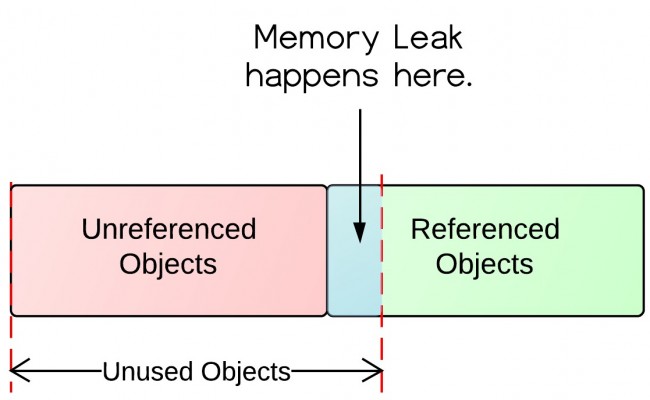
One of the most significant advantages of Java is its memory management. You simply create objects and Java Garbage Collector takes care of allocating and freeing memory. However, the situation is not as simple as that, because memory leaks frequently occur in Java applications.

This process is called garbage collection and the corresponding piece of JVM is called a Garbage Collector or GC. ... Simplifying a bit, we can say that a memory leak in Java is a situation where some objects are not used by the application any more, but GC fails to recognize them as unused.

**What is Memory Leak?**

Definition of Memory Leak: objects are no longer being used by the application, but Garbage Collector cannot remove them because they are being referenced.

To understand this definition, we need to understand objects status in memory. The following diagram illustrates what is unused and what is unreferenced.

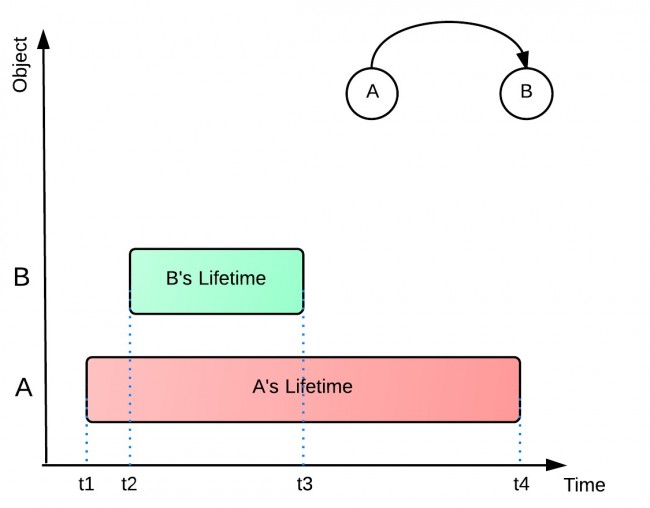


From the diagram, there are referenced objects and unreferenced objects. Unreferenced objects will be garbage collected, while referenced objects will not be garbage collected. Unreferenced objects are surely unused, because no other objects refer to it. However, unused objects are not all unreferenced. Some of them are being referenced! That's where the memory leaks come from.

Why Memory Leaks Happen?

Let's take a look at the following example and see why memory leaks happen. In the example below, object A refers to object B. A's lifetime (t1 - t4) is much longer than B's (t2 - t3). When B is no longer being used in the application, A still holds a reference to it. In this way, Garbage Collector can not remove B from memory. This would possibly cause out of memory problem, because if A does the same thing for more objects, then there would be a lot of objects that are uncollected and consume memory space.

It is also possible that B hold a bunch of references of other objects. Those objects referenced by B will not get collected either. All those unused objects will consume precious memory space.



**Why substring() method in JDK 6 can cause memory leaks?**

The substring(int beginIndex, int endIndex) method in JDK 6 and JDK 7 are different. Knowing the difference can help you better use them. For simplicity reasons, in the following substring() represent the substring(int beginIndex, int endIndex) method.

1. What substring() does?

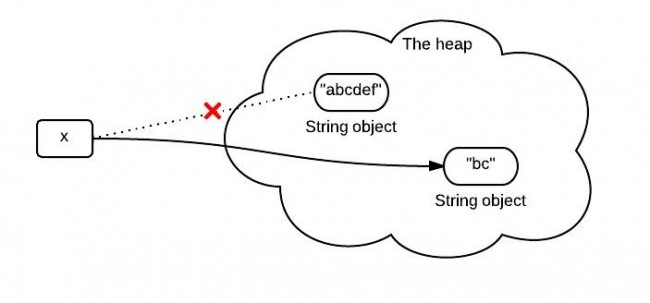
The substring(int beginIndex, int endIndex) method returns a string that starts with beginIndex and ends with endIndex-1.

|  |
| --- |
| String x = "abcdef";  x = x.substring(1,3);  System.out.println(x); |

Output: bc

2. What happens when substring() is called?

You may know that because x is immutable, when x is assigned with the result of x.substring(1,3), it points to a totally new string like the following:

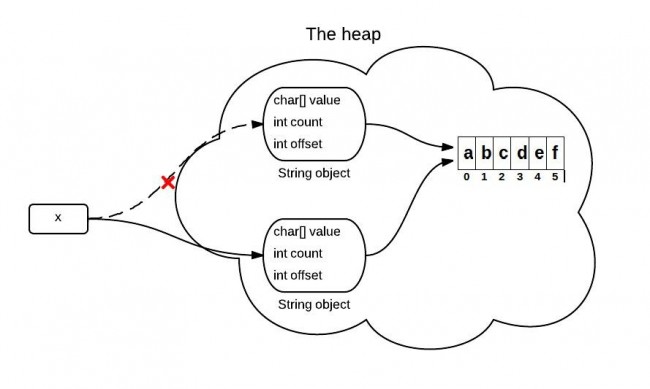


However, this diagram is not exactly right or it represents what really happens in the heap. What really happens when substring() is called is different between JDK 6 and JDK 7.

3. substring() in JDK 6

String is supported by a char array. In JDK 6, the String class contains 3 fields: char value[], int offset, int count. They are used to store real character array, the first index of the array, the number of characters in the String.

When the substring() method is called, it creates a new string, but the string's value still points to the same array in the heap. The difference between the two Strings is their count and offset values.



The following code is simplified and only contains the key point for explain this problem.

|  |
| --- |
| //JDK 6  String(int offset, int count, char value[]) {  this.value = value;  this.offset = offset;  this.count = count;  }    public String substring(int beginIndex, int endIndex) {  //check boundary  return new String(offset + beginIndex, endIndex - beginIndex, value);  } |

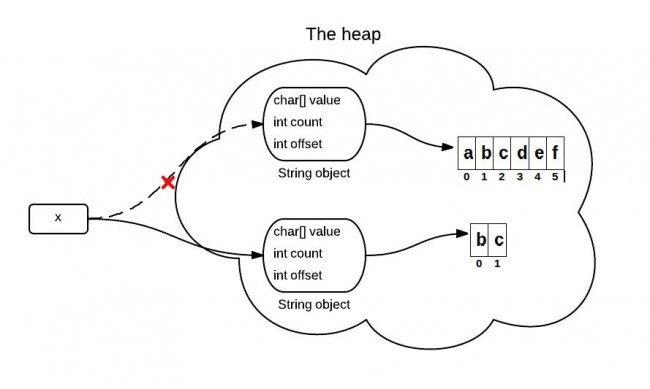
**4. A problem caused by substring() in JDK 6**

If you have a VERY long string, but you only need a small part each time by using substring(). This will cause a performance problem since you need only a small part, you keep the whole thing. For JDK 6, the solution is using the following, which will make it point to a real substring:

|  |
| --- |
| x = x.substring(x, y) + "" |

5. Substring() in JDK 7

This is improved in JDK 7. In JDK 7, the substring () method actually create a new array in the heap.



|  |
| --- |
| //JDK 7  public String(char value[], int offset, int count) {  //check boundary  this.value = Arrays.copyOfRange(value, offset, offset + count);  }    public String substring(int beginIndex, int endIndex) {  //check boundary  int subLen = endIndex - beginIndex;  return new String(value, beginIndex, subLen);  } |

**How to Prevent Memory Leaks?**

The following are some quick hands-on tips for preventing memory leaks.

Pay attention to Collection classes, such as HashMap, ArrayList, etc., as they are common places to find memory leaks. When they are declared static, their life time is the same as the life time of the application.

Pay attention to event listeners and callbacks. A memory leak may occur if a listener is registered but not unregistered when the class is not being used any longer.

"If a class manages its own memory, the programmer should be alert for memory leaks."[1] Often times member variables of an object that point to other objects need to be null out.

**What is a Thread dump?**

A thread dump is a snapshot of the state of all threads that are part of the process. The state of each thread is presented with a so called stack trace, which shows the contents of a thread's stack. Some of the threads belong to the Java application you are running, while others are JVM internal threads.

**[How will you take thread dump in Java? How will you analyze Thread dump?](http://www.fromdev.com/2008/05/java-threading-questions.html" \l "how-will-you-take-thread-dump-in-java--how-will-you-analyze-thread-dump-" \o "How will you take thread dump in Java? How will you analyze Thread dump?)**

A Thread Dump is a complete list of active threads. A java thread dump is a way of finding out what each thread in the JVM is doing at a particular point of time. This is especially useful when your java application seems to have some performance issues. Thread dump will help you to find out which thread is causing this. There are several ways to take thread dumps from a JVM. It is highly recommended to take more than 1 thread dump and analyze the results based on it. Follow below steps to take thread dump of a java processes.

Step 1   
  
On UNIX, Linux and Mac OSX Environment run below command:   
  
 ps -el | grep java   
  
On Windows:   
  
Press Ctrl+Shift+Esc to open the task manager and find the PID of the java process 

Step 2:   
  
Use jstack command to print the Java stack traces for a given Java process PID   
  
 jstack [PID]   
  
More details of jstack command can be found here: [JSTACK Command Manual](http://docs.oracle.com/javase/1.5.0/docs/tooldocs/share/jstack.html)

[How can I trace whether the application has a thread leak?](http://www.fromdev.com/2008/05/java-threading-questions.html#how-can-i-trace-whether-the-application-has-a-thread-leak-)

* If an application has thread leak then with time it will have too many unused threads. Try to find out what type of threads is leaking out. This can be done using following ways, Give unique and descriptive names to the threads created in application. –
* Add log entry in all thread at various entry and exit points in threads. Change debugging config levels (debug, info, error etc) and analyze log messages.
* When you find the class that is leaking out threads check how new threads are instantiated and how they're closed.
* Make sure the thread is guaranteed to close properly by doing following - Handling all Exceptions properly.
* Releasing all resources (e.g. connections, files etc) before it closes.

**How to Avoid Thread Interference or How to Achieve Thread Safeness?**

* By declaring the method as synchronized.
* By declaring the variables as final.
* By declaring the variable as volatile.
* By creating the immutable objects.
* By using Atomic operations.
* By restricting the access to same object by multiple threads.

**[What is thread pool? Why should we use thread pools?](http://www.fromdev.com/2008/05/java-threading-questions.html" \l "what-is-thread-pool--why-should-we-use-thread-pools-" \o "What is thread pool? Why should we use thread pools?)**

* A thread pool is a collection of threads on which task can be scheduled. Instead of creating a new thread for each task, you can have one of the threads from the thread pool pulled out of the pool and assigned to the task. When the thread is finished with the task, it adds itself back to the pool and waits for another assignment. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Below are key reasons to use a Thread Pool
* Using thread pools minimizes the JVM overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and de-allocating many thread objects creates a significant memory management overhead.
* You have control over the maximum number of tasks that are being processed in parallel (= number of threads in the pool).
* Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

**[Can two threads call two different synchronized instance methods of an Object?](http://www.fromdev.com/2008/05/java-threading-questions.html" \l "can-two-threads-call-two-different-synchronized-instance-methods-of-an-object-" \o "Can two threads call two different synchronized instance methods of an Object?)**

No. If a object has synchronized instance methods then the Object itself is used a lock object for controlling the synchronization. Therefore all other instance methods need to wait until previous method call is completed. See the below sample code which demonstrate it very clearly. The Class Common has 2 methods called synchronizedMetohd1() and synchronizedMethod2() MyThread class is calling both the methods.

[**Can we synchronize the run method? If yes then what will be the behavior?**](http://www.fromdev.com/2008/05/java-threading-questions.html#can-we-synchronize-the-run-method--if-yes-then-what-will-be-the-behavior-)

Yes, the run method of a runnable class can be synchronized. If you make run method synchronized then the lock on runnable object will be occupied before executing the run method. In case we start multiple threads using the same runnable object in the constructor of the Thread then it would work. But until the 1st thread ends the 2nd thread cannot start and until the 2nd thread ends the next cannot start as all the threads depend on lock on same object.

**What is Volatile variable in Java?**

volatile variable in Java is a special variable which is used to signal threads, a compiler that this particular variables value are going to be updated by multiple threads inside Java application. By making a variable volatile using the volatile keyword in Java, application programmer ensures that its value should always be read from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html)and thread should not use cached value of that variable from their own stack. With the introduction of Java memory model from Java 5 onwards along with introduction of [CountDownLatch](http://javarevisited.blogspot.sg/2012/07/countdownlatch-example-in-java.html), [CyclicBarrier](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html), [Semaphore](http://javarevisited.blogspot.sg/2012/05/counting-semaphore-example-in-java-5.html)and [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html), volatile variable also guarantees "happens-before" relationship, which means not only another thread has visibility of latest value of volatile variable but also all the variable is seen by the thread which has updated value of volatile variable before these threads sees it.

**What is volatile variable and when to use it is always a**[**popular Java threading question**](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html)**.**  
  
The volatile keyword can only be applied to a variable; it cannot be applied to class or method. Using volatile keyword along with class and method is a compiler error.

A volatile is also referred as modifier in Java.

**When to use Volatile variable in Java?**

1) Any variable which is shared between multiple threads should be made variable, in order to ensure that all thread must see the latest value of the volatile variable.  
  
2) A signal to compiler and JIT to ensure that compiler does not change ordering or volatile variable and moves them out of synchronized context.  
  
3) You want to save the cost of synchronization as volatile variables are less expensive than synchronization.  
  
**What are the different ways to create an object in Java?**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| There are many ways to create an object in java. They are:  By new keyword  By newInstance() method  By clone() method  By factory method etc.  Java String intern() method  A pool of strings, initially empty, is maintained privately by the class String. When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.  public class InternExample{  public static void main(String args[]){  String s1=new String("hello");  String s2="hello";  String s3=s1.intern();//returns string from pool, now it will be same as s2  System.out.println(s1==s2);//false because reference is different  System.out.println(s2==s3);//true because reference is same  }}  Difference between method overloading and method overriding in java?   |  |  | | --- | --- | | Method Overloading | Method Overriding | | Definition | When a class has more than one method with same name but with different arguments, then we call it as method overloading. | When a super class method is modified in the sub class, then we call this as method overriding. | | Method Signature | Overloaded methods must have different method signatures.  That means they should differ at least in any one of these three things – Number of arguments,  Types of arguments  And order of arguments. But, they must have same name. | Overridden methods must have same method signature. I.e. you must not change the method name, types of arguments, number of arguments and order of arguments while overriding a super class method. | | Return Types | Overloaded methods can have same or different return types. | The return type of the overridden method must be compatible with that of super class method. That means if super class method has primitive type as its return type, then it must be overridden with same return type. If super class method has derived type as its return type then it must be overridden with same type or its sub class type. | | Visibility(private, public, protected and default) | Overloaded methods can have same visibility or different visibility. | While overriding a super class method either you can keep the same visibility or you can increase the visibility. But you can’t reduce it. | | Static Context | Overloaded methods can be static or not static. It does not affect the method overloading. | You can’t override a static method. | | Binding | Binding between method call and method definition happens at compile time (Static Binding). | Binding between method call and method definition happens at run time (Dynamic Binding). | | Polymorphism | It shows static polymorphism. | It shows dynamic polymorphism. | | Private methods | Private methods can be overloaded. | Private methods can’t be overridden. | | Final Methods | Final methods can be overloaded. | Final methods can’t be overridden. | | Class Requirement | For method overloading, only one class is required. I.e. Method overloading happens within a class. | For method overriding, two classes are required – super class and sub class. That means method overriding happens between two classes. |    What are the rules to be followed while overriding a method?  There are 5 main rules you should kept in mind while overriding a method. They are,  a) Name of the method must be same as that of super class method.  b) Return type of overridden method must be compatible with the method being overridden. i.e if a method has primitive type as it’s return type then it must be overridden with primitive type only and if a method has derived type as it’s return type then it must be overridden with same type or it’s sub class types.  c) You must not reduce the visibility of a method while overriding.  d) You must not change parameter list of a method while overriding.  e) You cannot increase the scope of exceptions while overriding a method with throws clause.  Abstraction :  Yes, In the computer science, Abstraction is used to separate ideas from their implementation. Abstraction in java is used to define only ideas in one class so that the idea can be implemented by its sub classes according to their requirements.   1. Abstract classes and abstract methods are declared using ‘abstract‘ keyword. We can’t create objects to those classes which are declared as abstract. But, we can create objects to sub classes of abstract class, provided they must implement abstract methods. 2. The methods which are not implemented or which don’t have definitions must be declared with ‘abstract’ keyword and the class which contains it must be also declared as abstract. 3. It is not compulsory that abstract class must have abstract methods. It may or may not have abstract methods. But the class which has at least one abstract method must be declared as abstract. 4. You can’t create objects to abstract class even though it does not contain any abstract methods. 5. Abstract Class can be a combination of concrete and abstract methods. 6. Any class extending an abstract class must implement all abstract methods. If it does not implement, it must be declared as abstract. 7. Inside abstract class, we can keep any number of constructors. If you are not keeping any constructors, then compiler will keep default constructor. 8. Abstract methods cannot be private. Because, abstract methods must be implemented somehow in the sub classes. If you declare them as private, then you can’t use them outside the class. 9. Constructors and fields cannot be declared as abstract. 10. Abstract methods cannot be static.  [Interfaces In Java](http://javaconceptoftheday.com/interfaces-in-java/) Interfaces in java are very much similar to abstract classes but interfaces contain only abstract methods (you can refer to them as only ideas). Abstract classes may contain both abstract methods as well as concrete methods. But interfaces must contain only abstract methods. Concrete methods are not allowed in interfaces. Therefore, Interfaces show 100% abstractness.  Interfaces are declared with keyword ‘interface‘ and interfaces are implemented by the class using ‘implements‘ keyword.  Interfaces should contain only abstract methods. Interfaces should not contain a single concrete method.  Interface can have two types of members.  1) Fields     2) Abstract Methods.  By default, Every field of an interface is public, static and final (we will discuss about final keyword Later). You can’t use any other modifiers other than these three for a field of an interface. |
| You can’t change the value of a field once they are initialized. Because they are static and final. Therefore, sometimes fields are called as Constants. (We will discuss this feature in detail while covering ‘final’ keyword)  By default, All methods of an interface are public and abstract.  Like classes, for every interface .class file will be generated after compilation  While implementing any interface methods inside a class, that method must be declared as public. Because, according to [method overriding](http://javaconceptoftheday.com/method-overriding-java/) rule, you can’t reduce visibility of super class method. By default, every member of an interface is public and while implementing you should not reduce this visibility.  By default, Interface itself is not public but by default interface itself is abstract like below,  [SIB](http://javaconceptoftheday.com/static-members-java/) – Static Initialization Block and [IIB](http://javaconceptoftheday.com/instance-initialization-block-in-java/) – Instance Initialization Block are not allowed in interfaces.  As we all know that, any class in java cannot extend more than one class. But class can implement more than one interfaces. This is how multiple inheritance is implemented in java. |

# Differences between Static Binding and Dynamic Binding In Java?

The above findings can be summarized like below.

|  |  |
| --- | --- |
| Static Binding | Dynamic Binding |
| It is a binding that happens at compile time. | It is a binding that happens at run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| It is also called early binding because binding happens during compilation. | It is also called late binding because binding happens at run time. |
| Method overloading is the best example of static binding. | Method overriding is the best example of dynamic binding. |
| Private, static and final methods show static binding. Because, they cannot be overridden. | Other than private, static and final methods show dynamic binding. Because, they can be overridden. |

Difference between String and StringBuffer?

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

|  |  |  |
| --- | --- | --- |
| No. | String | StringBuffer |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you concat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

Difference between StringBuffer and StringBuilder?

There are many differences between StringBuffer and StringBuilder. A list of differences between StringBuffer and StringBuilder are given below:

|  |  |  |
| --- | --- | --- |
| No. | StringBuffer | StringBuilder |
| 1) | StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is less efficient than StringBuilder. | StringBuilder is more efficient than StringBuffer. |

## Does an interface extend Object class in java.?

You may have come across this question while reading about interfaces in java. You may also know that only classes in java are inherited from java.lang.Object class. Interfaces in java don’t inherit from Object class. They don’t have default parent like classes in java. But, following two cases may surprise you.

If an interface does not extend Object class, then why we can call methods of Object class on interface variable like below.

**If an interface does not extend Object class, then why the methods of Object class are visible in interface?**

This is because, for every public method in Object class, there is an implicit abstract and public method declared in every interface which does not have direct super interfaces. This is the standard Java Language Specification which states like this,

“If an interface has no direct super interfaces, then the interface implicitly declares a public abstract member method m with signature s, return type r, and throws clause corresponding to each public instance method m with signature s, return type r, and throws clause t declared in Object, unless a method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.”

**NoClassDefFoundError vs. ClassNotFoundException?**

In Java, both ClassNotFoundException and NoClassDefFoundError occur when a particular class is not found at run time. But, they occur at different scenarios. ClassNotFoundException is an exception which occurs when you try to load a class at run time using Class.forName() or loadClass() methods and mentioned classes are not found in the classpath. On the other hand, NoClassDefFoundError is an error which occurs when a particular class is present at compile time but it was missing at run time.

Difference Between ClassNotFoundException Vs NoClassDefFoundError In Java :

|  |  |
| --- | --- |
| ClassNotFoundException | NoClassDefFoundError |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the class path. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass () and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when class path is not updated with required JAR files. | It occurs when required class definition is missing at run time. |

**Difference between Exception and Error?**

Both java.lang.Error and java.lang.Exception classes are sub classes of java.lang.Throwable class, but there exist some significant differences between them. java.lang.Error class represents the errors which are mainly caused by the environment in which application is running. For example, OutOfMemoryError occurs when JVM runs out of memory or StackOverflowError occurs when stack overflows.

Where as java.lang.Exception class represents the exceptions which are mainly caused by the application itself. For example, NullPointerException occurs when an application tries to access null object or ClassCastException occurs when an application tries to cast incompatible class types. In this article, we will discuss the differences between Error and Exception in java.

`

|  |  |
| --- | --- |
| **Errors** | **Exceptions** |
| Errors in java are of type java.lang.Error. | Exceptions in java are of type java.lang.Exception. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions :  SQLException,IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |

**What is System.out in Java?**

In System.out, out is an instance of PrintStream. It is a static member variable in System class. This is called standard output stream, connected to console.

**Purpose of garbage collection.**

The garbage collection process is to identify the objects which are no longer referenced or needed by a program so that their resources can be reclaimed and reused. These identified objects will be discarded.

**Java Heap Space**

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

**Java Stack Memory**

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method. Stack memory size is very less compared to Heap memory.

**Difference between Java Heap Space and Stack Memory?**

Based on the above explanations, we can easily conclude following differences between Heap and Stack memory.

Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.

Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.

Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at [Java Garbage Collection](http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java). Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.

We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory.

We can use -Xss to define the stack memory size.

When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.

Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

**Abstraction**:

* Abstraction is "To represent the essential feature without representing the back ground details."
* Abstraction lets you focus on what the object does instead of how it does it.
* Abstraction provides you a generalized view of your classes or object by providing relevant information.
* Abstraction is the process of hiding the working style of an object, and showing the information of an object in understandable manner.

**Encapsulation**:

* Wrapping up data member and method together into a single unit (i.e. Class) is called Encapsulation.
* Encapsulation is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object.
* Encapsulation is like your bag in which you can keep your pen, book etc. It means this is the property of encapsulating members and functions.

**class Bag**

**{**

**book;**

**pen;**

**ReadBook();**

**}**

* Encapsulation means hiding the internal details of an object, i.e. how an object does something.
* Encapsulation prevents clients from seeing its inside view, where the behaviour of the abstraction is implemented.
* Encapsulation is a technique used to protect the information in an object from the other object.
* Hide the data for security such as making the variables as private, and expose the property to access the private data which would be public.
* So, when you access the property you can validate the data and set it.

**Abstraction/Encapsulation**

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| 1. Abstraction solves the problem in the design level. | 1. Encapsulation solves the problem in the implementation level. |
| 2. Abstraction is used for hiding the unwanted data and giving relevant data. | 2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world. |
| 3. Abstraction lets you focus on what the object does instead of how it does it | 3. Encapsulation means hiding the internal details or mechanics of how an object does something. |
| 4. Abstraction- Outer layout, used in terms of design.  For Example:-   Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number. | 4. Encapsulation- Inner layout, used in terms of implementation.  For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits. |

**Q) What is difference between polymorphism and inheritance?**

* Inheritance defines parent-child relationship between two classes, polymorphism take advantage of that relationship to add dynamic behavior in your code.
* Inheritance helps in code reusability by allowing child class to inherit behavior from the parent class. On the other hand Polymorphism allows Child to redefine already defined behavior inside parent class. Without Polymorphism it's not possible for a Child to execute its own behavior while represented by a Parent reference variable, but with Polymorphism he can do that.
* **Java doesn't allow multiple inheritances of classes, but allows**[**multiple inheritance of Interface**](http://java-questions.com/keyConcepts-interview-questions.html#diamond-problem)**, which is actually required to implement Polymorphism**. For example a Class can be Runnable, Comparator and Serializable at same time, because all three are interfaces. This makes them to pass around in code e.g. you can pass instance of this class to a method which accepts Serializable, or to Collections.sort() which accepts a Comparator.
* Both Polymorphism and Inheritance allow Object oriented programs to evolve. For example, by using Inheritance you can define new user types in an Authentication System and by using Polymorphism you can take advantage of already written authentication code. Since, Inheritance guarantees minimum base class behavior, a method depending upon super class or super interface can still accept object of base class and can authenticate it.

**What is Encapsulation?**

The encapsulation is achieved by combining the methods and attribute into a class. The class acts like a container encapsulating the properties. The users are exposed mainly public methods. The idea behind is to hide how things work and just exposing the requests a user can do.

**What is Association?**

Association is a relationship where all object have their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

**What is Aggregation?**

Aggregation is a specialize form of Association where all object have their own lifecycle but there is ownership and child object cannot belong to another parent object. Let's take an example of Department and teacher. A single teacher cannot belong to multiple departments, but if we delete the department, teacher object will not destroy. We can think about "has-a" relationship.

**What is Composition?**

Composition is again specialize form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. Child object does not have their lifecycle and if parent object deletes all child object will also be deleted. Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room cannot belong to two different house if we delete the house, room will automatically delete.

**Shallow Copy Vs Deep Copy in Java?**

Below is the list of differences between shallow copy and deep copy in java.

|  |  |
| --- | --- |
| Shallow Copy | Deep Copy |
| Cloned Object and original object are not 100% disjoint. | Cloned Object and original object are 100% disjoint. |
| Any changes made to cloned object will be reflected in original object or vice versa. | Any changes made to cloned object will not be reflected in original object or vice versa. |
| Default version of clone method creates the shallow copy of an object. | To create the deep copy of an object, you have to override clone method. |
| Shallow copy is preferred if an object has only primitive fields. | Deep copy is preferred if an object has references to other objects as fields. |
| Shallow copy is fast and also less expensive. | Deep copy is slow and very expensive. |

**Why is String immutable in Java?**

1. String Pool  
 When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object. If string is not immutable, changing the string with one reference will lead to the wrong value for the other references.  
2. To Cache its Hashcode  
 if string is not immutable, one can change its hashcode and hence not fit to be cached.  
3. Security  
 String is widely used as parameter for many java classes, e.g. network connection, opening files, etc. Making it mutable might possess threats due to interception by the other code segment.

**[What is immutable object in Java? Can you change values of a immutable object?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "what-is-immutable-object-in-java--can-you-change-values-of-a-immutable-object-" \o "What is immutable object in Java? Can you change values of a immutable object?)**

A Java object is considered immutable when its state cannot change after it is created. Use of immutable objects is widely accepted as a sound strategy for creating simple, reliable code. Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state. **java.lang.String** and **java.lang.Integer** classes are the Examples of immutable objects from the Java Development Kit. Immutable objects simplify your program due to following characteristics:

* Immutable objects are simple to use test and construct.
* Immutable objects are automatically thread-safe.
* Immutable objects do not require a copy constructor.
* Immutable objects do not require an implementation of clone.
* Immutable objects allow hash Code to use lazy initialization, and to cache its return value.
* Immutable objects do not need to be copied defensively when used as a field.
* Immutable objects are good Map keys and Set elements (Since state of these objects must not change while stored in a collection).
* Immutable objects have their class invariant established once upon construction, and it never needs to be checked again.
* Immutable objects always have "failure atomicity" (a term used by Joshua Bloch): if an immutable object throws an exception, it's never left in an undesirable or indeterminate state.

**[How to create a immutable object in Java? Does all property of immutable object needs to be final?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "how-to-create-a-immutable-object-in-java--does-all-property-of-immutable-object-needs-to-be-final-" \o "How to create a immutable object in Java? Does all property of immutable object needs to be final?)**

To create a object immutable You need to make the class final and all its member final so that once objects gets created no one can modify its state. You can achieve same functionality by making member as non final but private and not modifying them except in constructor. Also its NOT necessary to have all the properties final since you can achieve same functionality by making member as non final but private and not modifying them except in constructor.

[**What is difference between String, StringBuffer and StringBuilder? When to use them?**](http://www.fromdev.com/2012/02/java-interview-question-answer.html#what-is-difference-between-string--stringbuffer-and-stringbuilder--when-to-use-them-)

The main difference between the three most commonly used String classes as follows.

->StringBuffer and StringBuilder objects are mutable whereas String class objects are immutable.

->StringBuffer class implementation is synchronized while StringBuilder class is not synchronized.

Concatenation operator "+" is internally implemented by Java using either StringBuffer or StringBuilder.

Criteria to choose among String, StringBuffer and StringBuilder

-> If the Object value will not change in a scenario use String Class because a String object is immutable.

-> If the Object value can change and will only be modified from a single thread, use a StringBuilder because StringBuilder is unsynchronized(means faster).

-> If the Object value may change, and can be modified by multiple threads, use a StringBuffer because StringBuffer is thread safe(synchronized).

[**Why String class is final or immutable?**](http://www.fromdev.com/2012/02/java-interview-question-answer.html#why-string-class-is-final-or-immutable-)

It is very useful to have strings implemented as final or immutable objects. Below are some advantages of String Immutability in Java

* Immutable objects are thread-safe.
* Two threads can both work on an immutable object at the same time without any possibility of conflict.

Security:

* The system can pass on sensitive bits of read-only information without worrying that it will be altered.
* You can share duplicates by pointing them to a single instance.
* You can create substrings without copying. You just create a pointer into an existing base String guaranteed never to change. Immutability is the secret that makes Java substring implementation very fast.
* Immutable objects are good fit for becoming Hashtable keys. If you change the value of any object that is used as a hash table key without removing it and re-adding it you will lose the object mapping.
* Since String is immutable, inside each String is a char[] exactly the correct length. Unlike a String Builder there is no need for padding to allow for growth.
* If String were not final, you could create a subclass and have two strings that look alike when "seen as Strings", but that are actually different.

**How to create an immutable class?**

**To create an immutable class following steps should be followed:**

* Create a final class.
* Set the values of properties using constructor only.
* Make the properties of the class final and private
* Do not provide any setters for these properties.
* If the instance fields include references to mutable objects, don't allow those objects to be changed:
* Don't provide methods that modify the mutable objects.
* Don't share references to the mutable objects.
* Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

**What are the advantages of immutability?**

Immutable objects are automatically thread-safe; the overhead caused due to use of synchronization is avoided.

Once created the state of the immutable object cannot be changed so there is no possibility of them getting into an inconsistent state.

**The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state cannot be changed ever after construction.**

The best use of the immutable objects is as the keys of a map.

**Prevent Cloning**

To implement cloning, we have to implement java.lang.Cloneable interface and override clone () method from Object class. It is a good idea to prevent cloning in a singleton class. To prevent cloning on singleton object, let us explicitly throw CloneNotSupportedException exception in clone() method.

**Immutable objects are automatically thread-safe –true/false?**

True. Since the state of the immutable objects can not be changed once they are created they are automatically synchronized/thread-safe.

**Use of HashCode and Equals()?**

Object class provides two methods hashcode() and equals() to represent the identity of an object. It is a common convention that if one method is overridden then other should also be implemented.

Before explaining why, lets see what is the contract between these two methods hold. As per the Java API documentation:

Whenever hashcode is invoked on the same object more than once during an execution of a Java application, the hashcode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

If two objects are equal according to the equals(object) method, then calling the hashCode() method on each of the two objects must produce the same integer result.

It is NOT required that if two objects are unequal according to the equals(Java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

How to implement equals() method.

**Steps that need to be taken into consideration while implementing equals method.**

Use the == operator to check if the argument is a reference to this object. If so, return true. This is just a performance optimization, but one that is worth doing if the comparison is potentially expensive.

Use the instanceof operator to check if the argument has the correct type.

If not, return false. Typically, the correct type is the class in which the method occurs. Occasionally, it is some interface implemented by this class. Use an interface if the class implements an interface that refines the equals contract to permit comparisons across classes that implement the interface. Collection interfaces such as Set, List, Map, and Map.Entry have this property.

Cast the argument to the correct type. Because this cast was preceded by an instanceof test, it is guaranteed to succeed.

For each significant field in the class, checks if that field of the argument matches the corresponding field of this object.

If all these tests succeed, return true; otherwise, return false

**When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?**

The correct implementation of equals method for the String Helper class could be:

**Difference between throw and throws in Java?**

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| No. | throw | throws |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

**Difference Between Checked And Unchecked Exceptions :**

|  |  |
| --- | --- |
| Checked Exceptions | Unchecked Exceptions |
| They are known at compile time. | They are known at run time. |
| They are checked at compile time. | They are not checked at compile time. Because they occur only at run time. |
| These are compile time exceptions. | These are run time exceptions. |
| If  these exceptions are not handled properly in the application, they give compile time error. | If these exceptions are not handled properly, they don’t give compile time error. But application will be terminated prematurely at run time. |
| All sub classes of java.lang.Exception Class except sub classes of RunTimeException are checked exceptions. | All sub classes of RunTimeException and sub classes of java.lang.Error are unchecked exceptions. |

**Iterator vs Enumeration**

Between Enumeration and Iterator, Enumeration is older and its there from JDK1.0, while iterator was introduced later. Iterator can be used with ArrayList, HashSet and other collection classes.  Another similarity between Iterator and Enumeration in Java is that functionality of Enumeration interface is duplicated by the Iterator interface.

Only major difference between Enumeration and iterator is Iterator has a remove() method while Enumeration doesn't. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, where as by using Iterator we can manipulate the objects like adding and removing the objects from collection e.g. Arraylist.

**Also Iterator is more secure and safe as compared to Enumeration because it does not allow other thread to modify the collection object while some threads is iterating over it and throws ConcurrentModificationException**. This is by far most important fact for me for deciding between Iterator vs Enumeration in Java.

In Summary both Enumeration and Iterator will give successive elements, but Iterator is new and improved version where method names are shorter, and has new method called remove. Here is a short comparison:

Enumeration: hasMoreElement(),nextElement(),N/A,

Iterator : hasNext(),next(),remove()

So Enumeration is used whenever we want to make Collection objects as Read-only.

**Difference between final, finally and finalize**

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| No. | final | finally | finalize |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

|  |
| --- |
| **Exception Handling with Method Overriding in Java?**  There are many rules if we talk about method overriding with exception handling. The Rules are as follows:  If the super class method does not declare an exception  If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.  If the super class method declares an exception  If the super class method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |
| **Advantage of java inner classes?**  There are basically three advantages of inner classes in java. They are as follows:  1) Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.  2) Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.  3) Code Optimization: It requires less code to write. |
| |  |  | | --- | --- | | [Member Inner Class](http://www.javatpoint.com/member-inner-class) | A class created within class and outside method. | | [Anonymous Inner Class](http://www.javatpoint.com/anonymous-inner-class) | A class created for implementing interface or extending class. Its name is decided by the java compiler. | | [Local Inner Class](http://www.javatpoint.com/local-inner-class) | A class created within method. | | [Static Nested Class](http://www.javatpoint.com/static-nested-class) | A static class created within class. | | [Nested Interface](http://www.javatpoint.com/nested-interface) | An interface created within class or interface. | |

**There are 2 types of Nested Classes.?**

Static Nested Classes

Non-Static Nested Classes or Inner Classes

## Static Nested Classes In Java

* + If nested class is declared as static, then that nested class is called as **static nested class**
  + Static nested classes can contain both static and non-static members
  + We can access only static members of outer class inside a static nested class. We can’t access non-static members of outer class inside a static nested class.
  + We have seen that static methods can’t be abstract but static nested classes can be abstract.
  + Static nested class can be final.
  + Constructors and methods of nested classes can be overloaded.

## Non-Static Nested Classes In Java :

* + Nested classes which are declared as non-static or nested classes which can be accessed only though instantiating it’s outer class are called non-static nested classes. Non-static nested classes are also called as Inner Classes**.**

They are 3 types of Inner Classes in java.

* + **Member Inner Classes**

Member Inner Classes are non-static nested classes which are declared as non-static members of outer class.

Member inner classes must contain only non-static members. Static members are not allowed inside member inner classes.

You can declare a static field inside a member inner class if the field is final. And such field must be initialized at the time of declaration only. Remember, this rule is only for the fields not for the methods.

Member inner class may contain any number of IIB’s(Instantiation Initialization Block) but should not contain any SIB’s(Static Initialization Block).

We can access both static and non-static members of outer class inside a member inner class.

All members of outer class are accessible inside member inner class and all members of member inner class are accessible inside the outer class irrespective of their visibility.

Member inner classes can be abstract or can be final but not both.

* + **Local Inner Classes**

Local inner class in java is non-static nested class which is declared inside a method or a block.

Local inner classes must be defined inside a method or a block.

Local Inner Classes can’t be static. Because, local inner classes are nothing but local variables and local variables can’t be static.

Local inner classes can’t have static members. Only non-static members are allowed inside local inner classes. But local inner classes can contain static and final field.

To access members of local inner class, you must create an instance of it.

Local inner classes are local to a method or a block in which they are defined. i.e you can’t use local inner classes outside the method or block in which they are defined.

Only final local variables of methods or blocks containing local inner class can be used inside local inner class.

Local inner classes cannot be declared with access modifiers. i.e Local inner classes cannot be private, protected and public. But they can have private, public, protected and default members in them.

Local inner classes cannot be declared with access modifiers. i.e Local inner classes cannot be private, protected and public. But they can have private, public, protected and default members in them.

Local inner classes can be abstract or can be final but not both.

* + **Anonymous Inner classes**

Anonymous inner class, the name itself suggests that it is a class without a name. Anonymous inner class in java is an inner class or non-static nested class without a name.

Anonymous inner classes don’t have name. They are nameless.

You can create only one object to anonymous inner class. If you want to create another object, you have to write the whole class again.

When you are creating an anonymous inner class, you are actually creating a sub class to a class which needs to be modified. This sub class doesn’t have name and it is declared in another class. That’s why it is called Anonymous Inner Class.

While creating an anonymous inner class you are also creating an object to that subclass and it is referenced by super class reference variable. This also shows the polymorphism. Because, Super class reference variable can refer to super class object and also it’s sub class object.

Using anonymous inner class, you can implement both abstract classes and interfaces.

**Enum:**

**Enums** OR **Enum Types** OR **Enumeration types** are special data types which are used to represent similar kind of constants.

Enum types like classes can have fields, constructors and methods along with enum constants.

Enum constructors are private by default. Only private constructors are allowed in enum types. That’s why you can’t instantiate enum types using new operator.

Enum constants are created only once for the whole execution. All enum constants are created when you refer any enum constant first time in your code. While creating each enum constant, corresponding constructor is called.

Enum constants must be declared first ahead of fields, constructors and methods (if any).

All enum types extend java.lang.Enum class by default. As multiple inheritance is not supported in java, they can’t extend any other classes.

Enum types can implement any number of interfaces.

  Enum constants can have their own body called Constant Specific Body. In that body, you can define fields and methods. But, these methods and fields are visible within the Constant Specific Body in which they are defined.

Enum types are final by default. They can not be extended by any other types.

For every enum type written in a file, .class file will be generated after compilation.

Enum types can have any number of static initialization blocks as well as instance initialization blocks.

As java.lang.Enum class implements Comparable and Serializable interface, all enum types are Comparable andSerializable by default.

We can compare the enum constants using “==” operator.

You can retrieve the enum constants of any enum type using values() method. values() method returns an array of enum constants.

 Enums provide type-safety during compilation. That means you will get compile time error if you try to assign any other values other than the specified enum constants.

You can define enum types outside a class or inside a class but not inside a method or block.

ordinal() method is used get the order of an enum constant in an enum type.

 Enums are mostly used when you want to allow limited set of options which remain constant for whole execution and you know all possible options at compile time itself. For example, choices on a menu or options of a combobox.

**What if we call run() method directly instead start () method?**

|  |
| --- |
| Each thread starts in a separate call stack.  Invoking the run() method from main thread, the run () method goes onto the current call stack rather than at the beginning of a new call stack. |

**Daemon Thread in Java**

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically. There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the **jconsole** in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc. Points to remember for Daemon Thread in Java.

It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

Its life depends on user threads.

It is a low priority thread.

**Why JVM terminates the daemon thread if there is no user thread?**

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread? That is why JVM terminates the daemon thread if there is no user thread.

The java.lang.**Thread** class provides two methods for java daemon thread.

|  |  |  |  |
| --- | --- | --- | --- |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. | |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

**Java Garbage Collection**

In java, garbage means unreferenced objects. Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects. To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

**Advantage of Garbage Collection**

It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory. It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.

**How can an object be unreferenced?**

There are many ways:

By nulling the reference

By assigning a reference to another

By annonymous object etc.

**finalize() method**

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void finalize(){}

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

**gc() method**

The gc() method is used to invoke the garbage collector to perform cleanup processing.

The gc() is found in System and Runtime classes.

public static void gc(){}

Note: Garbage collection is performed by a daemon thread called Garbage Collector (GC).

This thread calls the finalize () method before object is garbage collected.

**The entire collection framework is divided into four interfaces.**

**1)** List  —> It handles sequential list of objects. ArrayList, Vector and LinkedList classes implement this interface.

2) Queue  —> It handles special list of objects in which elements are removed only from the head. LinkedList andPriorityQueue classes implement this interface.

3) Set  —> It handles list of objects which must contain unique element. This interface is implemented by HashSetand LinkedHashSet classes and extended by SortedSet interface which in turn, is implemented by TreeSet.

4) Map  —> This is the one interface in Collection Framework which is not inherited from Collection interface. It handles group of objects as Key/Value pairs. It is implemented by HashMap and HashTable classes and extended by SortedMap interface which in turn is implemented by TreeMap.

Three of above interfaces (List, Queue and Set) inherit from Collection interface. Although, Map is included in collection framework it does not inherit from Collection interface.

1. **List Interface:**

List Interface represents an ordered or sequential collection of objects. This interface has some methods which can be used to store and manipulate the ordered collection of objects. The classes which implement the List interface are called as Lists. ArrayList, Vector and LinkedList are some examples of lists. You have the control over where to insert an element and from where to remove an element in the list.

Here are some properties of lists.

Elements of the lists are ordered using Zero based index.

You can access the elements of lists using an integer index.

Elements can be inserted at a specific position using integer index. Any pre-existing elements at or beyond that position are shifted right.

Elements can be removed from a specific position. The elements beyond that position are shifted left.

A list may contain duplicate elements.

A list may contain multiple null elements.

List interface extends Collection interface. So, All 15 methods of Collection interface are inherited to List interface. Along with these methods, another 9 methods are included in the List interface to support the properties of lists.

1. [Advantages Of Using ArrayList Over Arrays](http://javaconceptoftheday.com/advantages-of-using-arraylist-over-arrays/)

Array and ArrayList are most used data types while developing any java applications. Both are used to store group of objects.

The drawbacks of arrays.

* + 1. Arrays are of fixed length. You can not change the size of the arrays once they are created.
    2. You can not accommodate an extra element in an array after they are created.
    3. Memory is allocated to an array during it’s creation only, much before the actual elements are added to it.

List down the advantages of using ArrayList over Arrays.

* + - * 1. You can define ArrayList as re-sizable array. Size of the ArrayList is not fixed. ArrayList can grow and shrink dynamically
        2. Elements can be inserted at or deleted from a particular position.
        3. ArrayList class has many methods to manipulate the stored objects.

d) If generics are not used, ArrayList can hold any type of objects.

1. Many are of the assumption that multiple insertion and removal operations on ArrayList will decrease the performance of an application. But, there will be no significant change in the performance of an application if you use ArrayList instead of arrays. Below example shows time taken to add 1000 string elements to ArrayList and array.
2. You can traverse an ArrayList in both the directions – forward and backward using ListIterator.
3. ArrayList can hold multiple null elements.
4. ArrayList can hold duplicate elements.
5. [**Collection Framework – The Queue Interface**](http://javaconceptoftheday.com/collection-framework-queue-interface/)

The Queue Interface extends Collection interface. It defines queue data structure which is normally First-In-First-Out. Queue is a data structure in which elements are added from one end and elements are deleted from another end. But, exception being the Priority Queue in which elements are removed from one end, but elements are added according to the order defined by the supplied comparator. Here is the hierarchy diagram of Queue interface.

**How Typical Queue Works?**

Queue is a data structure where elements are added from one end called tail of the queue and elements are removed from another end called head of the queue. Queue is also first-in-first-out type of data structure (except priority queue). That means an element which is inserted first will be the first element to be removed from the queue. You can’t add or get or set elements at an arbitrary position in the queues.

**Properties of Queue:**

1. Null elements are not allowed in the queue. If you try to insert null object into the queue, it throws NullPointerException.
2. Queue can have duplicate elements.
3. Unlike a normal list, queue is not random access. i.e you can’t set or insert or get elements at an arbitrary positions.
4. In most of cases, elements are inserted at one end called tail of the queue and elements are removed or retrieved from another end called head of the queue.
5. In the Queue Interface, there are two methods to obtain and remove the elements from the head of the queue. They are poll() and remove(). The difference between them is, poll() returns null if the queue is empty and remove() throws an exception if the queue is empty.
6. There are two methods in the Queue interface to obtain the elements but don’t remove. They are peek() and element(). peek() returns null if the queue is empty and element() throws an exception if the queue is empty.

## Methods Of Queue Interface:

Here are the methods of Queue interface. Some of the methods throw an exception if operation is not possible and some methods return a value (null or false) if operation is not possible.

|  |  |  |
| --- | --- | --- |
| Operation | Throws An Exception If operation is not possible | Returns null or false if operation is not possible |
| Add an element to the queue. | add() | offer() |
| Retrieve an element from the head of the queue. | element() | peek() |
| Retrieve And Remove an element from the head of the queue. | remove() | poll() |

1. [Collection Framework – The Deque Interface](http://javaconceptoftheday.com/collection-framework-deque-interface/)

The Deque is the short name for “Double Ended Queue”. As the name suggest, Deque is a linear collection of objects which supports insertion and removal of elements from both the ends. The Deque interface defines the methods needed to insert, retrieve and remove the elements from both the ends.

Properties Of Deque:

1. Unlike Queue, Deque can have null elements. But, it is recommended not to insert null elements as many methods return null to indicate Deque is empty.
2. Deque can have duplicate elements.
3. You can’t set or get or insert the elements at an arbitrary position of Deque. i.e Random access is not possible with the Deque.
4. You can use removeFirstOccurrenec(E e), removeLastOccurrence(E e) and remove(E e) methods to delete the elements from the Deque.
5. **Deque as Queue :**

As Deque interface extends Queue interface, it inherits all methods of Queue interface. So, you can use all those inherited methods to perform Queue operations. Along with them, methods defined in the Deque interface can also be used for Queue operations. Below is the list of Queue methods and their equivalent Deque methods.

|  |  |  |  |
| --- | --- | --- | --- |
| **Queue Methods** | | **Equivalent Deque Methods** | |
| add() | | 1. addLast() | |
| offer() | | 1. OfferLast() | |
| element() | | 1. getFirst() | |
| peek() | | 1. peekFirst() | |
| remove() | | 1. removeFirst() | |
| poll() | | 1. pollFirst() | |

**II. Deque as Stack:**

Deque interface has two more methods – pop() and push(). These two methods make Deque to function as a stack (Last-In-First-Out). Along with these two methods, you can also use addFirst(), peekFirst() and removeFirst() for stack operations. Below is the list of Stack methods and their equivalent methods of Deque.

|  |  |
| --- | --- |
| Stack Methods | Equivalent Deque Methods |
| push() | 1. addFirst() |
| pop() | 1. removeFirst() |
| peek() | 1. peekFirst() |

1. [**Collection Framework – The Set Interface**](http://javaconceptoftheday.com/collection-framework-set-interface/)**.**

The Set interface defines a set which contains a linear collection of objects with no duplicates. The Set interface extends Collection interface. Set interface does not have its own methods. All its methods are inherited from Collection interface. The only change that has been made to Set interface is that add() method will return false if you try to insert an element which is already present in the set**.**

## Properties of Set:

## Set contains only unique elements. It does not allow duplicates.

* + Set can contain only one null element.
  + Random access of elements is not possible.
  + Order of elements in a set is implementation dependent. HashSet elements are ordered on hash code of elements. TreeSet elements are ordered according to supplied Comparator (If no Comparator is supplied, elements will be placed in ascending order) and LinkedHashSet maintains insertion order.
  + Set interface contains only methods inherited from Collection interface. It does not have its own methods. But, applies restriction on methods so that duplicate elements are always avoided.
  + One more good thing about Set interface is that the stronger contract between equals() and hashCode() methods. According to this contract, you can compare two set instances of different implementation types (HashSet, TreeSet and LinkedHashSet).
  + Two set instances, irrespective of their implementation types, are said to be equal if they contain same elements.

1. [**Collection Framework – The SortedSet Interface**](http://javaconceptoftheday.com/collection-framework-sortedset-interface/)
   * The SortedSet interface extends Set interface. SortedSet is a set in which elements are placed according to supplied comparator. This Comparator is supplied while creating a SortedSet. If you don’t supply comparator, elements will be placed in ascending order. SortedSet interface defines 6 more methods along with the inherited methods from Set->Collection interfaces

## Properties of SortedSet Interface:

* + SortedSet cannot have null elements. If you try to insert null element, it gives NullPointerException at run time.
  + As SortedSet is a set, duplicate elements are not allowed.
  + SortedSet elements are sorted according to supplied Comparator. If you don’t mention any Comparator while creating a SortedSet, elements will be placed in ascending order.
  + Inserted elements must be of Comparable type and they must be mutually comparable.
  + You can retrieve first element and last elements of the SortedSet. You can’t access SortedSet elements randomly. i.e Random access is denied.
  + SortedSets returned by headSet(), tailSet() and subSet() methods are just views of the original set. So, changes in the returned set are reflected in the original set and vice versa.

|  |  |  |
| --- | --- | --- |
| SL NO. | **SortedSet Interface Methods** | **Description** |
| 1 | Comparator<? super E> comparator() | Returns Comparator used to order the elements. If no comparator is supplied, it returns null. |
| 2 | SortedSet<E> subSet(E fromElement, E toElement) | Returns a portion of this set whose elements range from ‘fromElement’ (Inclusive) and ‘toElement’ (Exclusive). |
| 3 | SortedSet<E> headSet(E toElement) | Returns a SortedSet whose elements are in the range from first element of the set (Inclusive) to ‘toElement’ (exclusive). |
| 4 | SortedSet<E> tailSet(E fromElement) | Returns a SortedSet whose elements are in the range from ‘fromElement’ (Inclusive) to last element of the set (exclusive). |
| 5 | E first() | Returns first element of the SortedSet. |
| 6 | E last() | Returns last element of the SortedSet |

1. [**Collection Framework – The NavigableSet Interface**](http://javaconceptoftheday.com/collection-framework-navigableset-interface/)**.**

The NavigableSet is a SortedSet with navigation facilities. The NavigableSet interface provides many methods through them you can easily find closest matches of any given element. It has the methods to find out less than, less than or equal to, greater than and greater than or equal of any element in a SortedSet.

**Properties of NavigableSet Interface:**

NavaigableSet can’t have null elements.

NavigableSet doesn’t support duplicate elements.

NavigableSet can be traversed and accessed in either ascending or descending order.

Methods subSet(), headSet() and tailSet() differ from SortedSet interface in taking additional arguments describing whether upper bound and lower bound are inclusive or exclusive.

1. [**Collection Framework – The ArrayList Class**](http://javaconceptoftheday.com/collection-framework-arraylist-class/)**.**

ArrayList, in simple terms, can be defined as re-sizable array. ArrayList is same like normal array but it can grow and shrink dynamically to hold any number of elements. ArrayList is a sequential collection of objects which increases or decreases in size as we add or delete the elements.

In ArrayList, elements are positioned according to Zero-based index. That means, elements are inserted from index 0. Default initial capacity of an ArrayList is 10. This capacity increases automatically as we add more elements to arraylist. You can also specify initial capacity of an ArrayList while creating it.

ArrayList class implements List interface and extends AbstractList. It also implements 3 marker interfaces –RandomAccess, Cloneable and Serializable.

## Properties Of ArrayList :

* Size of the ArrayList is not fixed. It can increase and decrease dynamically as we add or delete the elements.
* ArrayList can have any number of null elements.
* ArrayList can have duplicate elements.
* As ArrayList implements RandomAccess, you can get, set, insert and remove elements of the ArrayList from  any arbitrary position.
* When you insert an element in the middle of the ArrayList, the elements at the right side of that position are shifted one position right and when you delete an element, they will be shifted one position left. This feature of the ArrayList causes some performance issues as shifting of elements is time consuming if ArrayList has lots of elements.
* Elements are placed according to Zero-based index. That means, first element will be placed at index 0 and last element at index n-1, where ‘n’ is the size of the ArrayList.
* ArrayList is not synchronized. That means, multiple threads can use same ArrayList simultaneously.

[**Collection Framework – The Vector Class**](http://javaconceptoftheday.com/collection-framework-vector-class/)**.**

* The Vector Class is also dynamically grow-able and shrink-able collection of objects like an [ArrayList](http://javaconceptoftheday.com/collection-framework-arraylist-class/) class. But, the main difference between ArrayList and Vector is that Vector class is synchronized. That means, only one thread can enter into vector object at any moment of time.
* Vector class is preferred over ArrayList class when you are developing a multi threaded application. But, precautions need to be taken because vector may reduce the performance of your application as it is thread safety and only one thread is allowed to have object lock at any moment of time and remaining threads have to wait until a thread releases the object lock which is held by it. So, it is always recommended that if you don’t need thread safety environment, it is better to use ArrayList class than the Vector class.
* Vector class has same features as ArrayList. Vector class also extends AbstractList class and implements List interface. It also implements 3 marker interfaces – RandomAccess, Cloneable and Serializable.

## Properties Of Vector Class:

* The main feature of Vector class is that it is thread safety. All methods of Vector class are synchronized so that only one thread can execute them at any given time. This feature of Vector class is useful when you need thread safety code.
* Thread safety property of Vector class effects the performance of an application as it makes threads to wait for object lock.
* Capacity Increment: Capacity increment is an amount by which the capacity of the vector is automatically incremented whenever size of the vector exceeds it’s capacity. You can pass this capacity increment while creating a vector. If you don’t pass, capacity increment will be treated as zero and capacity of the vector will be doubled whenever size exceeds capacity.
* Unlike an ArrayList, you can set the size of the Vector manually. If the new size is greater than the current size, the new slots will be filled with null elements. If the new size is smaller than current size, then the extra elements will be discarded.
* You can traverse the vector using Enumeration object. Vector class has a method called elements() which returns an Enumeration object consisting of all elements of Vector.
* Vector class has separate methods to retrieve first and last element of vector object. You will not find these methods in ArrayList class. firstElement() retrieves first element and lastElement() method retrieves last element of the vector.

[**Why Not To Use Vector Class In Your Code?**](http://javaconceptoftheday.com/not-use-vector-class-code/)

Vector class is often considered as obsolete or “Due for Deprecation” by many experienced Java developers. They always recommend and advise not to use Vector class in your code. They prefer using ArrayList over Vector class. In this article, I have tried to list out some points regarding why not to use Vector class in your code.

1) You can achieve Thread Safety without Vector.

Vector class has only one advantage over ArrayList i.e it is thread safety. But, you can achieve thread safe ArrayList by using synchronizedList() method of Collections class. Below is the sample code.

public class MainClass

{

public static void main(String[] args)

{

ArrayList<Integer> list = new ArrayList<Integer>();

Collections.synchronizedList(list);

//It returns Synchronized list backed by original list.

}

}

2) Thread Safeness of Vector class is time consuming.

All methods of Vector class are synchronized. This makes each and every operation on Vector object thread safe. But, it is time consuming. Because, you need to acquire object lock for each operation you want to perform on vector object. Usually, you need set of operations to be synchronized not each and every operation. Isn’t make sense to take the object lock once, perform the operations you want and then release the lock when you are done. Why acquire the lock again and again for each operations?. This is the time consuming process and decreases the performance of your application.

3) Enumeration Vs Iterator

Vector class has a method which return Enumeration over the elements of Vector object. Although, Enumerations are faster than the Iterator, but it is not backed by the original collection. That means, any changes made to original collection does not reflect in Enumeration object. They ignore the modifications done during iteration. This may cause issues.

4) Is Vector class poorly designed?

Vector class combines two features – “Re-sizable Array” and “Synchronization“. This makes poor design. Because, if you need just “Re-sizable Array” and you use Vector class for that, you will get “synchronized Resizable Array” not just re-sizable array. This may reduce the performance of your application. Therefore, instead of using Vector class, always use ArrayList class. You will have re-sizable array and whenever you want to make it synchronized, use Collections.SynchronizedList().

[Java Collection Framework – The **LinkedList** Class](http://javaconceptoftheday.com/java-collection-framework-linkedlist-class/)

In general terms, LinkedList is a data structure where each element consist of three things. First one is the reference to previous element, second one is the actual value of the element and last one is the reference to next element.

The LinkedList class in Java is an implementation of doubly linked list which can be used both as a List as well asQueue. The LinkedList in java can have any type of elements including null and duplicates. Elements can be inserted and can be removed from both the ends and can be retrieved from any arbitrary position.

The LinkedList class extends AbstractSequentialList and implements List and Deque interfaces. It also implements 2 marker interfaces – Cloneable and Serializable.

Properties of LinkedList Class In Java

Elements in the LinkedList are called as Nodes. Where each node consist of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element.

Reference To Previous Element of first node and Reference To Next Element of last node are null as there will be no elements before the first node and after the last node.

You can insert the elements at both the ends and also in the middle of the LinkedList. Below is the list of methods for insertion operations

You can retrieve the elements form the head, from the middle and from the tail of the LinkedList. Below is the list of retrieval methods.

Insertion and removal operations in LinkedList are faster than the ArrayList. Because in LinkedList, there is no need to shift the elements after each insertion and removal. only references of next and previous elements need to be changed.

Retrieval of the elements is very slow in LinkedList as compared to ArrayList. Becaues in LinkedList, you have to traverse from beginning or end (whichever is closer to the element) to reach the element.

The LinkedList can be used as stack. It has the methods pop() and push() which make it to function as Stack.

The LinkedList can also be used as ArrayList, Queue, SIngle linked list and doubly linked list.

LinkedList can have multiple null elements.

LinkedList can have duplicate elements.

LinkedList class in Java is not of type Random Access. i.e the elements can not be accessed randomly. To access the given element, you have to traverse the LinkedList from beginning or end (whichever is closer to the element) to reach the given element.

[Java Collection Framework – The PriorityQueue Class](http://javaconceptoftheday.com/java-collection-framework-priorityqueue-class/)

The PriorityQueue is a queue in which elements are ordered according to specified Comparator. You have to specify this Comparator while creating a PriorityQueue itself. If no Comparator is specified, elements will be placed in their natural order. The PriorityQueue is a special type of queue because it is not a First-In-First-Out (FIFO) as in the normal queues. But, elements are placed according to supplied Comaparator.

The PriorityQueue does not allow null elements. Elements in the PriorityQueue must be of Comparable type, If you insert the elements which are not Comparable, you will get ClassCastException at run time.

PriorityQueue class extends AbstractQueue class which in turn implements Queue interface. PriorityQueue also implements one marker interface – java.io.Serializable interface.

Properties of PriorityQueue Class:

Elements in the PriorityQueue are ordered according to supplied Comparator. If Comparator is not supplied, elements will be placed in their natural order.

The PriorityQueue is unbounded. That means the capacity of the PriorityQueue increases automatically if the size exceeds capacity. But, how it grows is not specified.

The PriorityQueue can have duplicate elements but cannot have null elements.

All elements of the PriorityQueue must be of Comparable type.

Otherwise ClassCastException will be thrown at run time.

The head element of the PriorityQueue is always the least element and tail element is always the largest element according to specified Comparator.

The default initial capacity of PriorityQueue is 11.

You can retrieve the Comparator used to order the elements of the PriorityQueue using comparator() method.

PriorityQueue is not a thread safe.

[**Java Collection Framework – The HashSet Class**](http://javaconceptoftheday.com/java-collection-framework-hashset-class/)

* The HashSet class in Java is an implementation of Set interface. HashSet is a collection of objects which contains only unique elements. Duplicates are not allowed in HashSet. HashSet gives constant time performance for insertion, removal and retrieval operations. It allows only one null element.
* The HashSet internally uses HashMap to store the objects. The elements you insert in HashSet will be stored as keys of that HashMap object and their values will be a constant called PRESENT. This constant is defined as private static final Object PRESENT = new Object() in the source code of HashSet class.
* HashSet class extends AbstractSet class and implements Set interface. It also implements Cloneable and Serializable marker interfaces.

## Properties Of HashSet Class In Java

* HashSet class uses HashMap internally to store the objects. The keys of that HashMap object will be the elements of HashSet and their values will be a constant.
* HashSet does not allow duplicate elements. If you try to insert a duplicate element, older element will be overwritten.
* HashSet can have maximum one null element.
* HashSet doesn’t maintain any order. The order of the elements will be largely unpredictable. And it also doesn’t guarantee that order will remain constant over time.
* HashSet offers constant time performance for insertion, removal and retrieval operations
* HashSet class is not synchronized. If you want synchronized HashSet, use Collections.synchronizedSet()method.

# [Java Collection Framework – The LinkedHashSet Class](http://javaconceptoftheday.com/java-collection-framework-linkedhashset-class/)

* The LinkedHashSet in java is an ordered version of HashSet which internally maintains one doubly linked list running through it’s elements. This doubly linked list is responsible for maintaining the insertion order of the elements. Unlike HashSet which maintains no order, LinkedHashSet maintains insertion order of elements. i.e elements are placed in the order they are inserted. LinkedHashSet is recommended over HashSet if you want a unique collection of objects in an insertion order.
* The LinkedHashSet class extends HashSet class and implements Set interface. It also implements Cloneable and Serializable marker interfaces.
* Properties of LinkedHashSet Class In Java:
* LinkedHashSet internally uses LinkedHashMap to store it’s elements just like HashSet which internally uses HashMap to store it’s elements.
* LinkedHashSet maintains insertion order. This is the main difference between LinkedHashSet and HashSet.
* LinkedhashSet also gives constant time performance for insertion, removal and retrieval operations. The performance of LinkedHashSet is slightly less than the Hashset as it has to maintain doubly linked list internally to order it’s elements.
* Iterator returned by LinkedHashSet is fail-fast. i.e if the LinkedHashSet is modified at any time after the Iterator is created, it throws ConcurrentModificationException.
* LinkedHashSet doesn’t allow duplicate elements and allows only one null element.
* LinkedHashSet is not synchronized. To get the synchronized LinkedHashSet, useCollections.synchronizedSet() method.

[Java Collection Framework – The TreeSet Class](http://javaconceptoftheday.com/java-collection-framework-treeset-class/)

The TreeSet is another popular implementation of Set interface. We have seen other two implementations of Set interface –  [HashSet](http://javaconceptoftheday.com/java-collection-framework-hashset-class/) and [LinkedHashSet](http://javaconceptoftheday.com/java-collection-framework-linkedhashset-class/). HashSet doesn’t maintain any order where as LinkedHashSet maintains insertion order. The main difference between these two implementations and Treeset is, elements in TreeSet aresorted according to supplied Comparator. You need to supply this Comparator while creating a TreeSet itself. If you don’t pass any Comparator while creating a TreeSet, elements will be placed in their natural.

The TreeSet class in java is a direct implementation of NavigableSet interface which in turn extends SortedSet interface (which in turn extends Set interface).

## Properties Of TreeSet Class In Java

The elements in TreeSet are sorted according to specified Comparator. If no Comparator is specified, elements will be placed according to their natural ascending order.

Elements inserted in the TreeSet must be of Comparable type and elements must be mutually comparable. If the elements are not mutually comparable, you will get ClassCastException at run time

TreeSet does not allow even a single null element.

TreeSet is not synchronized. To get a synchronized TreeSet, use Collections.synchronizedSortedSet() method.

TreeSet gives performance of order log(n) for insertion, removal and retrieval operations.

Iterator returned by TreeSet is of fail-fast nature. That means, If TreeSet is modified after the creation of Iterator object, you will get ConcurrentModificationException.

TreeSet internally uses TreeMap to store it’s elements just like HashSet and LinkedHashSet which use HashMap and LinkedHashMap respectively to store their elements.

[J**ava Collection Framework – The Map Interface**](http://javaconceptoftheday.com/java-collection-framework-map-interface/)

The Map interface in java is one of the four top level interfaces of Java Collection Framework along with [List](http://javaconceptoftheday.com/collection-framework-list-interface/), [Set](http://javaconceptoftheday.com/collection-framework-set-interface/)and [Queue](http://javaconceptoftheday.com/collection-framework-queue-interface/) interfaces. But, unlike others, it doesn’t inherit from [Collection](http://javaconceptoftheday.com/collection-framework-collection-interface/) interface. Instead it starts it’s own interface hierarchy for maintaining the key-value associations. Map is an object of key-value pairs where each key is associated with a value. This interface is the replacement for ‘Dictionary‘ class which is an abstract class introduced in JDK 1.0.

HashMap, LinkedHashMap and TreeMap are three popular implementations of Map interface.

**Properties of Map Interface In Java:**

1) Map interface is a part of Java Collection Framework, but it doesn’t inherit Collection Interface.

2) Map interface stores the data as a key-value pairs where each key is associated with a value.

3) A map cannot have duplicate keys but can have duplicate values.

4) Each key at most must be associated with one value.

5) Each key-value pairs of the map are stored as Map.Entry objects. Map.Entry is an inner interface of Map interface.

6)The common implementations of Map interface are HashMap, LinkedHashMap and TreeMap.

7) Order of elements in map is implementation dependent. HashMap doesn’t maintain any order of elements.LinkedHashMap maintains insertion order of elements. Where as TreeMap places the elements according to supplied Comparator.

8) The Map interface provides three methods, which allows map’s contents to be viewed as a set of keys (keySet() method), collection of values (values() method), or set of key-value mappings (entrySet() method).

[What is the difference between Sorting performance of](http://www.fromdev.com/2008/05/java-collections-questions.html" \l "15) **[Arrays.sort](http://www.fromdev.com/2008/05/java-collections-questions.html" \l "15)** [() vs](http://www.fromdev.com/2008/05/java-collections-questions.html" \l "15) **[Collections.sort](http://www.fromdev.com/2008/05/java-collections-questions.html" \l "15)**[() ? Which one is faster? Which one to use and when?](http://www.fromdev.com/2008/05/java-collections-questions.html" \l "15)

Many developers are concerned about the performance difference between java.util.Array.sort() java.util.Collections.sort() methods. Both methods have same algorithm the only difference is type of input to them. Collections.sort() has a input as List so it does a translation of List to array and vice versa which is an additional step while sorting.   
  
So this should be used when you are trying to sort a list. Arrays.sort is for arrays so the sorting is done directly on the array. So clearly it should be used when you have a array available with you and you want to sort it.

[Java Collection Framework – The HashMap Class.](http://javaconceptoftheday.com/java-collection-framework-map-interface/)

The java.util.HashMap is a popular implementation of Map interface which holds the data as key-value pairs. HashMap extends AbstractMap class and implements Cloneable and Serializable interfaces. In this article, we will discuss about hierarchy of HashMap, properties of HashMap and some important methods of HashMap in java.

As already said, HashMap extends AbstractMap class and implements Cloneable and Serializable interfaces. AbstractMap is an abstract class which provides skeletal implementation of Map interface. Below is the hierarchy structure of java.util.HashMap class.

Properties Of HashMap In Java :

1) HashMap holds the data in the form of key-value pairs where each key is associated with one value.

2) HashMap doesn’t allow duplicate keys. But it can have duplicate values.

3) HashMap can have multiple null values and only one null key.

4) HashMap is not synchronized. To get the synchronized HashMap, use Collections.synchronizedMap () method.

5) HashMap maintains no order.

6) HashMap gives constant time performance for the operations like get() and put() methods.

7) Default initial capacity of HashMap is 16.

[**What is Load factor and Rehashing in Hashmap?**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)

**Load Factor:**

When the total number of items in hashmap goes on increasing keeping the default initial capacity of hashmap 16, At one point of time, hashmap performance will start degrading and need to increase buckets for improving performance.  
  
 Load Factor is a measure, which decides when exactly to increase the hashmap capacity (buckets) to maintain get and put operation complexity of O(1).

Default load factor of Hashmap is 0.75f (i.e 75% of current map size).

**Rehashing:**

Rehashing is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size hashmap when Load factor threshold is reached.  
  
When the number of items in map, crosses the Load factor limit at that time hashmap doubles its capacity and hashcode is re-calculated of already stored elements for even distribution of key-value pairs across new buckets.  
  
**Why Rehashing is required?**  
  
After doubling the capacity, what to do with the key-value pairs already present in buckets?

If we keep the existing key-value pairs as it is, then doubling the capacity may not help, because O(1) complexity will be achieved only if items are evenly distributed across all buckets.  
  
So for each existing key-value pairs, hashcode is calculated again with increased hashmap capacity as a parameter, which results in either placing the item in same bucket or in different bucket.

When the size of hashmap is changed, the process of re-calculating the hashcode of already placed key-value pair again is known as Rehashing.

Rehashing is done to distribute items across the new length hashmap, so that get and put operation time complexity remains O(1).  
  
NOTE: Hashmap maintain complexity of O(1) while inserting data in and getting data from hashmap, but for 12th key-value pair, put request will no longer be O(1), because as soon as map will realize that 12th element came in, that is 75% of map is filled. It will first double the bucket (array) capacity and then it will go for Rehash.  
 Rehashing requires re-computing hashcode of already placed 11 key-value pairs again and putting them at new index which requires time.

But overall time complexity provided by hashmap, which is O(1) for get and put operations, will amortize Rehashing process over long run.

**How To Convert HashMap To ArrayList In Java?**

As HashMap contains key-value pairs, there are three ways you can convert given HashMap to ArrayList. You can convert HashMap keys into ArrayList or you can convert HashMap values into ArrayList or you can convert key-value pairs into ArrayList.

a) **Conversion Of HashMap Keys Into ArrayList :**

For this, we use keySet() method of HashMap which returns the Set containing all keys of the HashMap. And then we pass this Set while constructing the ArrayList.

**//Creating a HashMap object**

**HashMap<String, String> map = new HashMap<String, String>();**

**//Getting Set of keys from HashMap**

**Set<String> keySet = map.keySet();**

**//Creating an ArrayList of keys by passing the keySet**

**ArrayList<String> listOfKeys = new ArrayList<String>(keySet);**

**b) Conversion Of HashMap Values Into ArrayList :**

For this, we use values() method of HashMap which returns the Collection containing all values of the HashMap. Then we use this Collection to create the ArrayList of values.

**//Creating a HashMap object**

**HashMap<String, String> map = new HashMap<String, String>();**

**//Getting Collection of values from HashMap**

**Collection<String> values = map.values();**

**//Creating an ArrayList of values**

**ArrayList<String> listOfValues = new ArrayList<String>(values);**

c) Conversion Of HashMap’s Key-Value Pairs Into ArrayList :

For this, we use entrySet() method of HashMap which returns the Set of Entry<K, V> objects where each Entry object represents one key-value pair. We pass this Set to create the ArrayList of key-value pairs.

**//Creating a HashMap object**

**HashMap<String, String> map = new HashMap<String, String>();**

**//Getting the Set of entries**

**Set<Entry<String, String>> entrySet = map.entrySet();**

**//Creating an ArrayList Of Entry objects**

**ArrayList<Entry<String, String>> listOfEntry = new ArrayList<Entry<String,String>>(entrySet);**

### How HashMap internally works in Java?

Hash Map is one of the most used collection, though it will be surprising to know that maps themselves are not collections because they don't implement Collection interface. However collection view of a map can be obtained using entrySet() method. To obtain a collection-view of the keys, keySet() method can be used.

Coming to the internal working of the HashMap, which is also a favourite [Java Collections interview question](http://netjs.blogspot.com/2015/11/java-collections-interview-questions.html), there are four things we should know about before going into the internals of how does HashMap work in Java -

HashMap works on the principal of hashing.

Map.Entry interface - This interface gives a map entry (key-value pair). HashMap in Java stores both key and value object, in bucket, as an object of Entry class which implements this nested interface Map.Entry.

hashCode() -HashMap provides put(key, value) for storing and get(key) method for retrieving Values from HashMap. When put() method is used to store (Key, Value) pair, HashMap implementation calls hashcode on Key object to calculate a hash that is used to find a bucket where Entry object will be stored. When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.

equals() - equals() method is used to compare objects for equality. In case of HashMap key object is used for comparison, also using equals() method Map knows how to handle hashing collision (hashing collision means more than one key having the same hash value, thus assigned to the same bucket. In that case objects are stored in a linked list, refer [figure](http://netjs.blogspot.in/2015/05/how-hashmap-internally-works-in-java.html#hashmap) for more clarity.   
Where hashCode method helps in finding the bucket where that key is stored, equals method helps in finding the right key as there may be more than one key-value pair stored in a single bucket.

\*\* Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0, table[1] as bucket1 and so on.

Refer [Overriding hashCode() and equals() method in Java](http://netjs.blogspot.com/2015/06/overriding-hashcode-and-equals-method.html) to know more about hashCode() and equals() method

How important it is to have a proper hash code and equals method can be seen through the help of the following program -

public class HashMapTest {

public static void main(String[] args) {

Map <Key, String> cityMap = new HashMap<Key, String>();

cityMap.put(new Key(1, "NY"),"New York City" );

cityMap.put(new Key(2, "ND"), "New Delhi");

cityMap.put(new Key(3, "NW"), "Newark");

cityMap.put(new Key(4, "NP"), "Newport");

System.out.println("size before iteration " + cityMap.size());

Iterator <Key> itr = cityMap.keySet().iterator();

while (itr.hasNext()){

System.out.println(cityMap.get(itr.next()));

}

System.out.println("size after iteration " + cityMap.size());

}

}

// This class' object is used as key

// in the HashMap

class Key{

int index;

String Name;

Key(int index, String Name){

this.index = index;

this.Name = Name;

}

@Override

// A very bad implementation of hashcode

// done here for illustrative purpose only

public int hashCode(){

return 5;

}

@Override

// A very bad implementation of equals

// done here for illustrative purpose only

public boolean equals(Object obj){

return true;

}

}

Output

size before iteration 1

Newport

size after iteration 1

Understanding the Code

Lets get through the code to see what is happening, this will also help in understanding how put works internally.

Notice that I am inserting 4 values in the [HashMap](http://netjs.blogspot.com/2015/05/how-to-loop-iterate-hash-map-in-java.html), still in the output it says size is 1 and iterating the map gives me the last inserted entry. Why is that?

Answer lies in, how hashCode() and equals() method are implemented for the key Class. Have a look at the hashCode() method of the class Key which always returns "5" and the equals() method which is always returning "true".

When a value is put into HashMap it calculates a hash using key object and for that it uses the hashCode() method of the key object class (or its parent class). Based on the calculated hash value HashMap implementation decides which bucket should store the particular Entry object.

In my code the hashCode() method of the key class always returns "5". This effectively means calculated hash value, is same for all the entries inserted in the HashMap. Thus all the entries are stored in the same bucket.

Second thing, a HashMap implementation does is to use equals() method to see if the key is equal to any of the already inserted keys (Recall that there may be more than one entry in the same bucket). Note that, with in a bucket key-value pair entries (Entry objects) are stored in a linked-list (Refer figure for more clarity). In case hash is same, but equals() returns false (which essentially means more than one key having the same hash or hash collision) Entry objects are stored, with in the same bucket, in a linked-list.

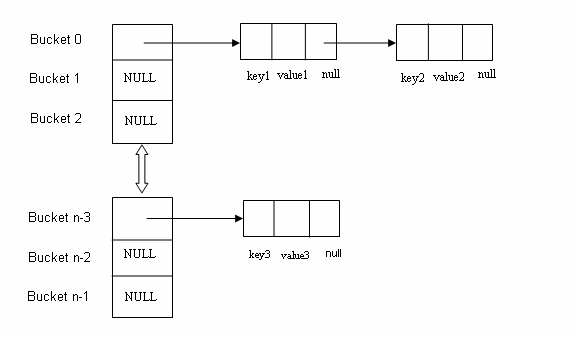
In my code, I am always returning true for equals() method so the HashMap implementation "thinks" that the keys are equal and overwrites the value. So, in a way using hashCode() and equals() I have "tricked" HashMap implementation to think that all the keys (even though different) are same, thus overwriting the values.

In a nutshell there are three scenarios in case of put() -

Using hashCode() method, hash value will be calculated. Using that hash it will be ascertained, in which bucket particular entry will be stored.

equals() method is used to find if such a key already exists in that bucket, if no then a new node is created with the map entry and stored within the same bucket. A linked-list is used to store those nodes.

If equals() method returns true, which means that the key already exists in the bucket. In that case, the new value will overwrite the old value for the matched key.

[](http://4.bp.blogspot.com/-x_U5Yjgsg8c/VVNgwvz7WNI/AAAAAAAAAHg/BreaAUMvJpc/s1600/hashMap+internal.png)

Pictorial representation of how Entry (key-value pair) objects will be stored in table array

How get() methods works internally

As we already know how Entry objects are stored in a bucket and what happens in the case of Hash Collision it is easy to understand what happens when key object is passed in the get method of the HashMap to retrieve a value.

Using the key again hash value will be calculated to determine the bucket where that Entry object is stored, in case there are more than one Entry object with in the same bucket stored as a linked-list equals() method will be used to find out the correct key. As soon as the matching key is found get() method will return the value object stored in the Entry object.

In case of null Key

As we know that HashMap also allows null, though there can only be one null key in HashMap. While storing the Entry object HashMap implementation checks if the key is null, in case key is null, it always map to bucket 0 as hash is not calculated for null keys.

HashMap changes in Java 8

Though HashMap implementation provides constant time performance O(1) for get() and put() method but that is in the ideal case when the Hash function distributes the objects evenly among the buckets.

But the performance may worsen in the case hashCode() used is not proper and there are lots of hash collisions. As we know now that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst case scenario the complexity becomes O(n).

To address this issue in Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree, this will improve the worst case performance from O(n) to O(log n).

Points to note -

HashMap works on the principal of hashing.

HashMap uses the hashCode() method to calculate a hash value. Hash value is calculated using the key object. This hash value is used to find the correct bucket where Entry object will be stored.

HashMap uses the equals() method to find the correct key whose value is to be retrieved in case of get() and to find if that key already exists or not in case of put().

Hashing collision means more than one key having the same hash value, in that case Entry objects are stored as a linked-list with in a same bucket.

With in a bucket values are stored as Entry objects which contain both key and value.

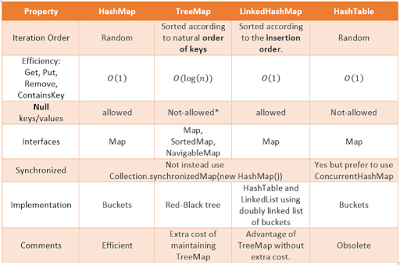
In Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached while storing values. This improves the worst case performance from O(n) to O(log n).

**Difference between HashMap and Hashtable in Java?**

Both HashMap and Hashtable implements Map interface but there is some significant difference between them which is important to remember before deciding whether to use HashMap or Hashtable in Java. Some of them are thread-safety, synchronization, and speed. here are those differences:

1. The HashMap class is roughly equivalent to Hashtable, except that it is non-synchronized and permits nulls. (HashMap allows null values as key and value whereas [Hashtable](http://javarevisited.blogspot.sg/2012/01/java-hashtable-example-tutorial-code.html)doesn't allow nulls).

2. One of the major differences between HashMap and Hashtable is that HashMap is non-synchronized whereas Hashtable is synchronized, which means Hashtable is thread-safe and can be shared between multiple threads but HashMap cannot be shared between multiple threads without proper synchronization. Java 5 introduces [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html)which is an alternative of Hashtable and provides better scalability than Hashtable in Java.

[](https://3.bp.blogspot.com/-iw2pMTU6pHo/VvP3HoW7XrI/AAAAAAAAFSQ/By53EkcfMGkoth857hnsc57XxYy8uwfAA/s1600/Difference+between+HashMap,+TreeMap,+LinkedHashMap+and+hashtable+in+Java.png)

3. Another significant difference between HashMap vs Hashtable is that Iterator in the HashMap is  a fail-fast iterator  while the enumerator for the Hashtable is not and throw ConcurrentModificationException if any other Thread modifies the map structurally  by adding or removing any element except Iterator's own remove() method. But this is not a guaranteed behavior and will be done by JVM on best effort. This is also an important [difference between Enumeration and Iterator in Java](http://javarevisited.blogspot.sg/2010/10/what-is-difference-between-enumeration.html).  
  
4. One more notable difference between Hashtable and HashMap is that because of thread-safety and synchronization Hashtable is much slower than HashMap if used in Single threaded environment. So if you don't need synchronization and HashMap are only used by one thread, it outperforms Hashtable in Java.  
  
5. HashMap does not guarantee that the order of the map will remain constant over time.

**HashMap and Hashtable : note on Some Important Terms**

1) Synchronized means only one Thread can modify a hash table at one point of time. Basically, it means that any thread before performing an update on a Hashtable will have to acquire a lock on the object while others will wait for the lock to be released.

2) Fail-safe is relevant from the context of iterators. If an [Iterator or ListIterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) has been created on a collection object and some other thread tries to modify the collection object "structurally", a concurrent modification exception will be thrown. It is possible for other threads though to invoke "set" method since it doesn't modify the collection "structurally". However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will be thrown.

**Difference between ConcurrentHashMap and Hashtable?**

So what is the difference between Hashtable and ConcurrentHashMap, both can be used in the multithreaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for a longer duration.

Since ConcurrentHashMap introduced the concept of segmentation, how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.  In Summary, ConcurrentHashMap only locked certain portion of Map while Hashtable locks full map while doing iteration. This will be clearer by looking at this diagram which explains the internal working of ConcurrentHashMap in Java.

**The difference between ConcurrentHashMap and Collections.synchronizedMap?**

ConcurrentHashMap is designed for concurrency and improve performance while HashMap which is non-synchronized by nature can be synchronized by applying a wrapper using synchronized Map. Here are some of the common differences between ConcurrentHashMap and synchronized map in Java  
  
ConcurrentHashMap does not allow null keys or null values while synchronized HashMap allows one null key.  
**Comparator vs Comparable**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Comparable** | **Comparator** |
| Sorting logic | Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects | Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc. |
| Implementation | Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id | Class whose objects to be sorted do not need to implement this interface. Some other class can implement this interface. E.g.-Country SortById Comparator class can implement Comparator interface to sort collection of country object by id |
| Sorting method | int compareTo(Object o1) This method compares this object with o1 object and returns  a integer. Its value has following meaning 1. positive – this object is greater than o1 2. zero – this object equals to o1 3. negative – this object is less than o1 | int compare(Object o1,Object o2) This method compares o1 and o2 objects. and returns  a integer. Its value has following meaning. 1. positive – o1 is greater than o2 2. zero – o1 equals to o2 3. negative – o1 is less than o1 |
| Calling method | Collections.sort(List) Here objects will be sorted on the basis of CompareTo method | Collections.sort(List, Comparator) Here objects will be sorted on the basis of Compare method in Comparator |
| Package | Java.lang.Comparable | Java.util.Comparator |

**What is Fail-fast in java?**

A fail-fast system is nothing but immediately report any failure that is likely to lead to failure. When a problem occurs, a fail-fast system fails immediately. In Java, we can find this behavior with iterators. In case, you have called iterator on a collection object, and another thread tries to modify the collection object, then concurrent modification exception will be thrown. This is called fail-fast.

**What is Fail-safe iterator?**

Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure. Any structural modification done to the iterator affects the copied data structure. So, original data structure remains structurally unchanged .Hence, no ConcurrentModificationException throws by the fail safe iterator.

Two issues associated with Fail Safe Iterator are :

1. Overhead of maintaining the copied data structure i.e memory.

2. Fail safe iterator does not guarantee that the data being read is the data currently in the original data structure.

Recap : Difference between Fail Fast Iterator and Fail Safe Iterator 

|  |  |  |
| --- | --- | --- |
|  | Fail Fast Iterator | Fail Safe Iterator |
| Throw ConcurrentModification Exception | Yes | No |
| Clone object | No | Yes |
| Memory Overhead | No | Yes |
| Examples | HashMap,Vector,ArrayList,HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

**What will happen if you put return statement or System.exit () on try or catch block? Will finally block execute?**

This is a very popular tricky Java question and it's tricky because many programmers think that no matter what, but the [finally block](http://java67.blogspot.com/2016/06/difference-between-final-vs-finally-vs-finalize-in-java.html) will always execute. This question challenge that concept by putting a return statement in the try or catch block or calling System.exit() from try or catch block. Answer of this tricky question in Java is that finally block will execute even if you put a return statement in the try block or catch block but finally block won't run if you call System.exit() from try or catch block. 

**Synchronization:**

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource. The synchronization is mainly used to

**To prevent thread interference,**

**To prevent consistency problem.**

There are two types of synchronization - Process Synchronization, Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

Mutual Exclusive- Synchronized method, Synchronized block, static synchronization.

Cooperation (Inter-thread communication in java)

**Concept of Lock in Java**

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations

**Static synchronization**

If you make any static method as synchronized, the lock will be on the class not on object.

**What is Dead Lock?**

When two or more threads are waiting for each other to release lock and get stuck for infinite time, situation is called deadlock. It will only happen in case of multitasking.

**[How to find a deadlock has occurred in Java? How to detect a Deadlock in Java?](http://www.fromdev.com/2008/05/java-threading-questions.html" \l "how-to-find-a-deadlock-has-occurred-in-java-" \o "How to find a deadlock has occurred in Java?)**

Earlier versions of Java had no mechanism to handle/detect deadlock. Since JDK 1.5 there are some powerful methods added in the java.lang.management package to diagnose and detect deadlocks. The java.lang.management.ThreadMXBean interface is management interface for the thread system of the Java virtual machine. It has two methods which can leverage to detect deadlock in a Java application.

findMonitorDeadlockedThreads() - This method can be used to detect cycles of threads that are in deadlock waiting to acquire object monitors. It returns an array of thread IDs that are deadlocked waiting on monitor.

findDeadlockedThreads() - It returns an array of thread IDs that are deadlocked waiting on monitor or ownable synchronizers.

[**How to Avoid the Deadlock in Java**](http://javaconceptoftheday.com/avoid-the-deadlock-java/)**?**

Deadlock is a dangerous condition, if it happens , it will bring the whole application to complete halt. So, extra care need to be taken to avoid the deadlock. Followings are some tips that can be used to avoid the deadlock in java.

Try to avoid nested synchronized blocks. Nested synchronized blocks makes a thread to acquire another lock while it is already holding one lock. This may create the deadlock if another thread wants the same lock which is currently held by this thread.

**Lock Ordering :**

If you needed nested synchronized blocks at any cost, then make sure that threads acquire the needed locks in some predefined order. For example, If there are three threads t1, t2 and t3 running concurrently and they needed locks A, B and C in the following manner,

[?](http://javaconceptoftheday.com/avoid-the-deadlock-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Thread t1 :          Lock A          Lock B  Thread t2 :          Lock A          Lock C  Thread t3 :          Lock A          Lock B          Lock C |

In the above scenario, t1 needs A and B locks, t2 needs A and C locks and t3 needs A, B and C locks. If you define an order to acquire the locks like, Lock A must be acquired before Lock B and Lock B must be acquired before Lock c, then deadlock never occurs in the above case.

If you define such lock ordering, then thread t2 never acquire lock C and t3 never acquire lock B and lock C until they got lock A. They will wait for lock A until it is released by t1. After lock A is released by t1, any one of these threads will acquire lock A on the priority basis and finishes their task. Other thread which is waiting for lock A, will never try to acquire remaining locks.

By defining such lock ordering, you can avoid the deadlock.

**Lock Timeout:**

Another deadlock preventive tip is to specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time. Such method of specifying time to acquire the lock is called lock timeout.

Lock the code where it is actually needed. For example, If you want only some part of the method to be thread safety, then lock only that part not the whole method.

**Method vs Block level synchronization**

Scope of lock is reduced by Synchronized block. As scope of lock is inversely proportional to performance, that's why it's always better to lock only critical section of the code. The best example of using synchronized block is double checked locking in Singleton pattern where instead of locking whole getInstance() method we only lock critical section of code which is used to create Singleton instance. This improves performance drastically because locking is only required one or two times.

In case of synchronized block, thread acquires lock when they enter synchronized block and release when they leave synchronized block. On the other hand in case of synchronized method, lock is acquired by thread when it enters method and released when it leaves method, either normally or by throwing Exception.

**Difference between "implements Runnable" and "extends Thread" in Java?**

### 1) Multiple Inheritance Limitation

As you know, Java doesn’t support multiple inheritance. A class in java can extend only one class. If you extend Thread class, then your class will not be able to extend any other class. This will limit your class to thread behavior. If you implement Runnable interface, then you will have an option for your class to extend any other class and inherit behaviors from other class also.

### 2) Overhead Of Additional Methods

If you extend Thread class, all methods of Thread class will be inheriting to your class which you may not need. This will cause additional overhead. You can remove this overhead by implementing Runnable interface.

### 3) Logical Separation Of Task From The Runner

If you implement Runnable interface, it will separate actual task from the runner. Runnable interface represents only the task and you can pass this task to any type of runner, either a thread or any executors.

### 4) Best Object Oriented Design Practice

In object oriented programming, extending a class means modifying or improving the existing class. If you are not improving the class, then it is not a good practice to extend it. So, implementing Runnable will be the best object oriented design practice.

### 5) Loosely Coupled Vs Tightly coupled

“Implements Runnable” makes your code loosely coupled. Because it separates the task from the runner. “Extends Thread” will make your code tightly coupled. Because, single class will act as both task container as well as runner.

### 6) Reusability

Implementing Runnable improves the reusability of your code. Because, Runnable contains only the task and you can use it wherever and whenever you want.

### 7) Specialization Vs Generalization

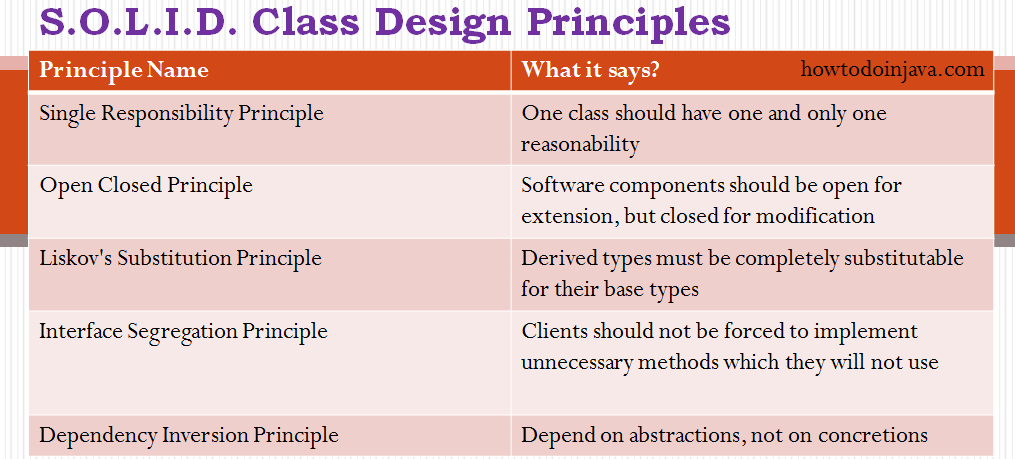
“Extends Thread” gives more specialized code. Because, it defines the thread specific task. Where as “Implements Runnable” gives more generalized version of the task applicable to many threads.

### 8) Maintenance

“Implements Runnable” will make your code easily maintainable as it separates the task from the runner. If you want to modify the task at any time, you can do so easily without disturbing the runner.

**Extends Thread Vs Implements Runnable In Java :**

|  |  |
| --- | --- |
| Implements Runnable | Extends Thread |
| You can extend any other class. | You can’t extend any other class. |
| No overhead of additional methods . | Overhead of additional methods from Thread class. |
| Separates the task from the runner. | Doesn’t separate the task from the runner. |
| Best object oriented programming practice. | Not a good object oriented programming practice. |
| Loosely coupled. | Tightly coupled. |
| Improves the reusability of the code. | Doesn’t improve the reusability of the code. |
| More generalized task. | Thread specific task. |
| Maintenance of the code will be easy. | Maintenance of the code will be time consuming. |



**Some of the benefits of collections framework are;**

Reduced development effort by using core collection classes rather than implementing our own collection classes. Code quality is enhanced with the use of well tested collections framework classes. Reduced effort for code maintenance by using collection classes shipped with JDK.

Reusability and Interoperability

**Describe, in general, how java's garbage collector works?**

The Java runtime environment deletes objects when it determines that they are no longer being used. This process is known as garbage collection. The Java runtime environment supports a garbage collector that periodically frees the memory used by objects that are no longer needed. The Java garbage collector is a mark-sweep garbage collector that scans Java's dynamic memory areas for objects, marking those that are referenced. After all possible paths to objects are investigated, those objects that are not marked (i.e. are not referenced) are known to be garbage and are collected.

**What is synchronization and why is it important in Java?**

|  |
| --- |
| W**hat is synchronization and why is it important? Describe synchronization in respect to multithreading.**  Synchronization is the process of allowing threads to execute one after another. Synchronization control the access the multiple threads to a shared resources. Without synchronization of threads, one thread can modify a shared variable while another thread can update the same shared variable, which leads to significant errors.  **What is synchronization and why is it important?**  Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method's object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor. |

**Join Method in Thread:**

Join method belongs to Thread object and not a static member. When join is invoked an a thread instance, this thread will tell currently executing thread to wait till the Joining thread completes Join is used in the situations when a task which should be completed before the current task is going to finish.

Just to give another example of how Join works, say we have a slow train on the track moving and another fast train should take the same track, so, we make way to the fast train and tell the slow train to

join once after the fast one is left.

Yield Method in Thread**:** Yield is a Static method.

Yield tells the currently executing thread to give a chance to the threads that have equal priority in the Thread Pool. There is no guarantee that Yield will make the currently executing thread to runnable state immediately.

Remember an important point that yield method does not make the thread to go to wait or Blocked state. It can only make a thread from Running State to Runnable State.

Life cycle of a Thread (Thread States)

[Life cycle of a thread](http://www.javatpoint.com/life-cycle-of-a-thread) : [New](http://www.javatpoint.com/life-cycle-of-a-thread#threadstatenew),[Runnable](http://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunnable),[Running](http://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunning) ,[Non-Runnable (Blocked)](http://www.javatpoint.com/life-cycle-of-a-thread#threadstateblocked) ,[Terminated](http://www.javatpoint.com/life-cycle-of-a-thread#threadstateterminated)

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

|  |
| --- |
| 1) **New**  The thread is in new state if you create an instance of Thread class but before the invocation  of start() method. |

2) **Runnable**

The thread is in runnable state after invocation of start () method, but the thread scheduler has not selected it to be the running thread.

3) **Running**

The thread is in running state if the thread scheduler has selected it.

4) **Non-Runnable (Blocked)**

This is the state when the thread is still alive, but is currently not eligible to run.

5) **Terminated**

A thread is in terminated or dead state when its run() method exits.

Some Things-To-Remember About wait(), notify() and notifyAll() :

If a thread calls notify() method and more than one threads are waiting for the object lock, then only one thread will be notified randomly.

When a thread calls notifyAll() method on an object, it notifies all the threads which are waiting for this object lock. But, only one thread will acquire this object lock depending upon priority.

When you call sleep() method on a thread, thread goes to sleep with holding the object lock with it. But, if you call wait() method, thread releases the object lock and goes for sleep. This is the main difference between wait() and sleep() methods.

wait(), notify() and notifyAll() are final methods of java.lang.Object class not java.lang.Thread class.

wait(), notify() and notifyAll() – all these three methods throw IllegalMonitorStateException if the calling thread does not owns the object lock.

wait() method is overloaded in Object class. There are two more wait() methods available in Object class. They are,

public final void wait(long timeOut)  —>  This makes current thread to wait until any other thread calls notify() or notifyAll() on this object or specified time(milli seconds) has elapsed.

public final void wait(long timeOut, int nanos)  —>  This makes current thread to wait until any other thread calls notify() or notifyAll() on this object or specified time(milli seconds + nano seconds) has elapsed.

Interrupt ( non static method)

Thread interruption in java is a mechanism in which a thread which is either sleeping or waiting can be made to stop sleeping or waiting. The whole thread interruption mechanism depends on an internal flag called interrupt status. The initial value of this flag for any thread is false. When you call interrupt() method on a thread, interrupt status of that thread will be set to true. When a thread throws InterruptedException, this status will be set to false again. Remember, InterruptedException is thrown when a thread is interrupted while it is sleeping or waiting. Many methods of Thread class like sleep(), wait(), join() throw InterruptedException.

[7 Things Every Java Programmer Should Know About Threads In Java](http://javaconceptoftheday.com/7-things-every-java-programmer-should-know-about-threads-in-java/)?

1. If you start a thread that is already started, you will get java.lang.IllegalThreadStateException at run time. There will be no compilation errors.
2. Exception is thread wise not execution wise. i.e exception effects the thread in which it occurs. Other threads will execute normally. In the below example, exception occurs in thread t1. only this thread will be terminated abruptly. Thread t2 will continue to execute it’s task.
3. As we all know that start() method internally calls run() method. What happens when you call run() method directly?. When you call run() method of a thread directly, calling thread will execute the task defined in the run() method. For example, in the below program main thread is calling run() method of thread t. In this case, main thread will execute run() method not thread t.
4. Which one is better way to implement threads in java. Is it using Thread class or using Runnable interface?. It is the most confusing question for a java developer. I am of opinion that when multiple threads need to execute same task, then use Runnable interface. If multiple threads need to execute different tasks, then go for Thread class.
5. Setting the priority to a thread is not effective as we thought. Setting Priority of a thread is just an advice to OS not an instruction. It is up to OS to consider this advice.
6. A thread is a permanent member of a thread group to which it joins during creation. You can’t move a thread to a new group after creating it.

Differences Between wait () and sleep() Methods In Java?

|  |  |
| --- | --- |
| wait() | sleep() |
| The thread which calls wait() method releases the lock it holds. | The thread which calls sleep() method doesn’t release the lock it holds. |
| The thread regains the lock after other threads call either notify() or notifyAll() methods on the same lock. | No question of regaining the lock as thread doesn’t release the lock. |
| wait() method must be called within the synchronized block. | sleep() method can be called within or outside the synchronized block. |
| wait() method is a member of java.lang.Object class. | sleep() method is a member of java.lang.Thread class. |
| wait() method is always called on objects. | sleep() method is always called on threads. |
| wait() is a non-static method of Object class. | sleep() is a static method of Thread class. |
| Waiting threads can be woken up by other threads by calling notify() or notifyAll() methods. | Sleeping threads can not be woken up by other threads. If done so, thread will throw InterruptedException. |
| To call wait() method, thread must have object lock. | To call sleep() method, thread need not to have object lock. |

What is the difference between ArrayList and LinkedList ?

Ans. Underlying data structure for ArrayList is Array whereas LinkedList is the linked list and hence have following differences -  
  
1. ArrayList needs continuous memory locations and hence need to be moved to a bigger space if new elements are to be added to a filled array which is not required for LinkedList.  
  
2. Removal and Insertion at specific place in ArrayList requires moving all elements and hence leads to O(n) insertions and removal whereas its constant O(1) for LinkedList.  
  
3. Random access using index in ArrayList is faster than LinkedList which requires traversing the complete list through references.  
  
4. Though Linear Search takes Similar Time for both, Binary Search using LinkedList requires creating new Model called Binary Search Tree which is slower but offers constant time insertion and deletion.  
  
5. For a set of integers you want to sort using quicksort, it's probably faster to use an array; for a set of large structures you want to sort using selection sort, a linked list will be faster.

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| 1) ArrayList internally uses dynamic array to store the elements. | LinkedList internally uses doubly linked list to store the elements. |
| 2) Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can act as a list only because it implements List only. | LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |
| 4) ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |

**When to use ArrayList or LinkedList ?**

1. Adding new elements is pretty fast for either type of list. Inserting element to nth location in arraylist and to first location in linkedlist takes O(1).
2. For the ArrayList, doing random lookup using "get" is faster O(1), but for LinkedList O(n), it's slow. It's slow because there's no efficient way to index into the middle of a linked list. Linkedlist lookup always start from 1st location.
3. When removing elements, using ArrayList is slow. This is because all remaining elements in the underlying array of Object instances must be shifted down for each remove operation. But LinkedList is fast, because deletion can be done simply by changing a couple of links.

So an ArrayList works best for cases where you're doing random access on the list and a LinkedList works better if you're doing a lot of editing in the middle of the list.

**HashSet Vs TreeSet Vs LinkedHashSet In Java ?**

Even though, **HashSet**, **LinkedHashSet** and **TreeSet** are all implementations of Set interface, there are some differences exist between them. In this article, I have tried to list out the differences between HashSet, LinkedHashSet and TreeSet in java. They also have some similarities between them. We will also discuss them at the end.

Differences Between HashSet, LinkedHashSet and TreeSet In Java :

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

**Similarities between HashSet, LinkedHashSet and TreeSet In Java?**

All three doesn’t allow duplicate elements.

All three are not synchronized.

All three are Cloneable and Serializable.

Iterator returned by all three is fail-fast in nature. i.e You will get ConcurrentModificationException if they are modified after the creation of Iterator object.

**[What changes are compatible and incompatible to the mechanism of java Serialization?](http://www.fromdev.com/2012/06/15-java-serialization-interview.html" \l "what-changes-are-compatible-and-incompatible-to-the-mechanism-of-java-serialization-" \o "What changes are compatible and incompatible to the mechanism of java Serialization?)**

This is one of a difficult and tricky questions and answering this correctly would mean you are an expert in Java Serialization concept. In an already serialized object, the most challenging task is to change the structure of a class when a new field is added or removed. As per the specifications of Java Serialization, addition of any method or field is considered to be a compatible change whereas changing of class hierarchy or non-implementation of Serializable interface is considered to be a non-compatible change. You can go through the Java serialization specification for the extensive list of compatible and non-compatible changes. If a serialized object need to be compatible with an older version, it is necessary that the newer version follows some rules for compatible and incompatible changes. A compatible change to the implementing class is one that can be applied to a new version of the class, which still keeps the object stream compatible with older version of same class.

**Some Simple Examples of compatible changes are:**

* Addition of a new field or class will not affect serialization, since any new data in the stream is simply ignored by older versions. The newly added field will be set to its default values when the object of an older version of the class is unmarshaled.
* The access modifiers change (like private, public, protected or default) is compatible since they are not reflected in the serialized object stream.
* Changing a transient field to a non-transient field is compatible change since it is similar to adding a field.
* Changing a static field to a non-static field is compatible change since it is also similar to adding a field.

**Some Simple Examples of incompatible changes are:**

* Changing implementation from Serializable to Externalizable interface cannot be done since this will result in the creation of an incompatible object stream.
* Deleting a existing Serializable fields will cause a problem.
* Changing a non-transient field to a transient field is incompatible change since it is similar to deleting a field.
* Changing a non-static field to a static field is incompatible change since it is also similar to deleting a field.
* Changing the type of a attribute within a class would be incompatible, since this would cause a failure when attempting to read and convert the original field into the new field.
* Changing the package of class is incompatible. Since the fully-qualified class name is written as part of the object byte stream.

Java serialization is one of the most commonly misunderstood areas.

Many developers still think it’s only used for saving objects on the file system.

## [Is it possible to customize the serialization process? How can we customize the Serialization process?](http://www.fromdev.com/2012/06/15-java-serialization-interview.html" \l "is-it-possible-to-customize-the-serialization-process--how-can-we-customize-the-serialization-process-" \o "Is it possible to customize the serialization process? How can we customize the Serialization process?)

Yes, the serialization process can be customized. When an object is serialized, objectOutputStream.writeObject (to save this object) is invoked and when an object is read, ObjectInputStream.readObject () is invoked. What most people do not know is that Java Virtual Machine provides you with an option to define these methods as per your needs. Once this is done, these two methods will be invoked by the JVM instead of the application of the default serialization process. Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

private void writeObject(java.io.ObjectOutputStream out)

throws IOException

private void readObject(java.io.ObjectInputStream in)

throws IOException, ClassNotFoundException;

private void readObjectNoData()

throws ObjectStreamException;

## [What are the alternatives to Serialization? If Serialization is not used, is it possible to persist or transfer an object using any other approach?](http://www.fromdev.com/2012/06/15-java-serialization-interview.html" \l "what-are-the-alternatives-to-serialization--if-serialization-is-not-used--is-it-possible-to-persist-or-transfer-an-object-using-any-other-approach-" \o "What are the alternatives to Serialization? If Serialization is not used, is it possible to persist or transfer an object using any other approach?)

In case, Serialization is not used, Java objects can be serialized by many ways, some of the popular methods are listed below:

* Saving object state to database, this is most common technique used by most applications. You can use ORM tools (e.g. hibernate) to save the objects in a database and read them from the database.
* Xml based data transfer is another popular mechanism, and a lot of XML based web services use this mechanism to transfer data over network. Also a lot of tools save XML files to persist data/configurations.
* JSON Data Transfer - is recently popular data transfer format. A lot of web services are being developed in JSON due to its small footprint and inherent integration with web browser due to JavaScript format.

## [When will you use Serializable or Externalizable interface? and why?](http://www.fromdev.com/2012/06/15-java-serialization-interview.html" \l "when-will-you-use-serializable-or-externalizable-interface--and-why-" \o "When will you use Serializable or Externalizable interface? and why?)

Most of the times when you want to do a selective attribute serialization you can use Serializable interface with transient modifier for variables not to be serialized. However, use of Externalizable interface can be really effective in cases when you have to serialize only some dynamically selected attributes of a large object.   
  
Lets take an example, Some times when you have a big Java object with hundreds of attributes and you want to serialize only a dozen dynamically selected attributes to keep the state of the object you should use Externalizable interface writeExternal method to selectively serialize the chosen attributes.  
  
In case you have small objects and you know that most or all attributes are required to be serialized then you should be fine with using Serializable interface and use of transient variable as appropriate.

**Serialization and SerialVersionUID?**

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is serializable if its class or any of its super classes implements either the java.io.Serializable interface or its sub interface, java.io.Externalizable.

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException.

A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:ANY-ACCESS-MODIFIER

**static final long serialVersionUID = 42L;**

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members.

**What is the use of hashcode in Java ?**

Ans. Hashcode is used for bucketing in Hash implementations like HashMap, HashTable, HashSet etc. The value received from hashcode() is used as bucket number for storing elements. This bucket number is the address of the element inside the set/map. when you do contains () then it will take the hashcode of the element, then look for the bucket where hashcode points to and if more than 1 element is found in the same bucket (multiple objects can have the same hashcode) then it uses the equals() method to evaluate if object are equal, and then decide if contain() is true or false, or decide if element could be added in the set or not.   
 **Difference between SAX and DOM Parser?**

A DOM (Document Object Model) parser creates a tree structure in memory from an input document whereas A SAX (Simple API for XML) parser does not create any internal structure.  
  
A SAX parser serves the client application always only with pieces of the document at any given time whereas A DOM parser always serves the client application with the entire document no matter how much is actually needed by the client.  
  
A SAX parser, however, is much more space efficient in case of a big input document whereas DOM parser is rich in functionality.  
  
Use a DOM Parser if you need to refer to different document areas before giving back the information. Use SAX is you just need unrelated nuclear information from different areas.  
  
Xerces, Crimson are SAX Parsers whereas XercesDOM, SunDOM, Oracle DOM are DOM parsers.

**[Is Java Pass by Reference or Pass by Value?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "is-java-pass-by-reference-or-pass-by-value-" \o "Is Java Pass by Reference or Pass by Value?)**

The Java Spec says that everything in Java is pass-by-value. There is no such thing as "pass-by-reference" in Java. The difficult thing can be to understand that Java passes "objects as references" passed by value.

[**What is OutOfMemoryError in java? How to deal with java.lang.OutOfMemeryError error?**](http://www.fromdev.com/2012/02/java-interview-question-answer.html#what-is-outofmemoryerror-in-java--how-to-deal-with-java-lang-outofmemeryerror--error-)

This Error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

Note: Its an Error (extends java.lang.Error) not Exception.

Two important types of OutOfMemoryError are often encountered

java.lang.OutOfMemoryError: Java heap space

The quick solution is to add these flags to JVM command line when Java runtime is started:

**-Xms1024m -Xmx1024m**

java.lang.OutOfMemoryError: PermGen space

The solution is to add these flags to JVM command line when Java runtime is started:

-**XX:+CMSClassUnloadingEnabled-XX:+CMSPermGenSweepingEnabled**

**Long Term Solution:** Increasing the Start/Max Heap size or changing Garbage Collection options may not always be a long term solution for your Out Of Memory Error problem. Best approach is to understand the memory needs of your program and ensure it uses memory wisely and does not have leaks. You can use a **Java memory profiler** to determine what methods in your program are allocating large number of objects and then determine if there is a way to make sure they are no longer referenced, or to not allocate them in the first place.

**[What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "what-is-the-use-of-the-finally-block--is-finally-block-in-java-guaranteed-to-be-called--when-finally-block-is-not-called-" \o "What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?)**

Finally is the block of code that executes always. The code in finally block will execute even if an exception is occurred. Finally block is NOT called in following conditions

If the JVM exits while the try or catch code is being executed, then the finally block may not execute. This may happen due to System.exit () call.

if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

If a exception is thrown in finally block and not handled then remaining code in finally block may not be executed.

**[Why there are two Date classes; one in java.util package and another in java.sql?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "why-there-are-two-date-classes--one-in-java-util-package-and-another-in-java-sql-" \o "Why there are two Date classes; one in java.util package and another in java.sql?)**

From the JavaDoc of java.sql.Date:

A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE value. A milliseconds value represents the number of milliseconds that have passed since January 1, 1970 00:00:00.000 GMT. To conform with the definition of SQL DATE, the millisecond values wrapped inside a java.sql.Date instance must be 'normalized' by setting the hours, minutes, seconds, and milliseconds to zero.

Explanation: A java.util.Date represents date and time of day, a java.sql.Date only represents a date (the complement of java.sql.Date is java.sql.Time, which only represents a time of day, but also extends java.util.Date).

**[What is Marker interface? How is it used in Java?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "what-is-marker-interface--how-is-it-used-in-java-" \o "What is Marker interface? How is it used in Java?)**

**The marker interface is a design pattern**, used with languages that provide run-time type information about objects. It provides a way to associate metadata with a class where the language does not have explicit support for such metadata. To use this pattern, a class implements a marker interface, and code that interact with instances of that class test for the existence of the interface. Whereas a typical interface specifies methods that an implementing class must support, a marker interface does not do so. The mere presence of such an interface indicates specific behavior on the part of the implementing class. There can be some hybrid interfaces, which act as markers and specify required methods, are possible but may prove confusing if improperly used. Java utilizes this pattern very well and the example interfaces are

**java.io.Serializable** - Serializability of a class is enabled by the class implementing the java.io.Serializable interface. The Java Classes that do not implement Serializable interface will not be able to serialize or deserialize their state. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

**java.rmi.Remote -** The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends **java.rmi.Remote** are available remotely.

**java.lang.Cloneable** - A class implements the Cloneable interface to indicate to the Object.clone () method that it is legal for that method to make a field-for-field copy of instances of that class. Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.

**javax.servlet.SingleThreadModel -** Ensures that servlets handle only one request at a time. This interface has no methods.

**java.util.EvenListener** - A tagging interface that all event listener interfaces must extend. The "instanceof" keyword in java can be used to test if an object is of a specified type. So this keyword in combination with Marker interface can be used to take different actions based on type of interface an object implements.

**[Why main() in java is declared as public static void main? What if the main method is declared as private?](http://www.fromdev.com/2012/02/java-interview-question-answer.html" \l "why-main---in-java-is-declared-as-public-static-void-main-" \o "Why main() in java is declared as public static void main?)**

Public - main method is called by JVM to run the method which is outside the scope of project therefore the access specifier has to be public to permit call from anywhere outside the application static - When the JVM makes are call to the main method there is not object existing for the class being called therefore it has to have static method to allow invocation from class. void - Java is platform independent language therefore if it will return some value then the value may mean different to different platforms so unlike C it cannot assume a behavior of returning value to the operating system. If main method is declared as private then - Program will compile properly but at run-time it will give "Main method not public." error.

**Why is it preferred to declare: List<String> list = new ArrayList<String>(); instead of ArrayList<String> = new ArrayList<String>();**

Ans) It is preferred because:

1. If later on code needs to be changed from ArrayList to Vector then only at the declaration place we can do that.
2. The most important one – If a function is declared such that it takes list. E.g void showDetails(List list);  
   When the parameter is declared as List to the function it can be called by passing any subclass of List like ArrayList, Vector, LinkedList making the function more flexible.

**Which data structure HashSet implements ?**

Ans) HashSet implements hashmap internally to store the data. The data passed to hashset is stored as key in hashmap with null as value.

**What is the difference between iterator access and index access?**

Index based access allow access of the element directly on the basis of index. The cursor of the data structure can directly goto the 'n' location and get the element. It does not traverse through n-1 elements.

In Iterator based access, the cursor has to traverse through each element to get the desired element. So to reach the 'n'th element it needs to traverse through n-1 elements.

Insertion,updation or deletion will be faster for iterator based access if the operations are performed on elements present in between the data structure.

Insertion,updation or deletion will be faster for index based access if the operations are performed on elements present at last of the data structure.

Traversal or search in index based data structure is faster.

ArrayList is index access and LinkedList is iterator access.

**How to sort a list in reverse order?**

To sort the elements in the reverse natural order of the strings, get a reverse Comparator from the Collections class with reverse Order(). Then, pass the reverse Comparator to the sort() method.

**List list = new ArrayList();**

**Comparator comp = Collections.reverseOrder();**

**Collections.sort(list, comp)**

**Can a null element be added to a Treeset or HashSet ?**

Ans) A null element can be added only if the set is of size 1 because when a second element is added then as per set defination a check is made to check duplicate value and comparison with null element will throw NullPointerException.  
HashSet is based on hashMap and can contain null element.

**How to sort list of strings - case insensitive?**

using Collections.sort(list, String.CASE\_INSENSITIVE\_ORDER);

**How to make a List (ArrayList, Vector,LinkedList) read only?**

A list implemenation can be made read only using **Collections.unmodifiableList(list)**.

This method returns a new list. If a user tries to perform add operation on the new list; **UnSupportedOperationException** is thrown.

**Which is faster to iterate LinkedHashSet or LinkedList?**

LinkedList.

**Arrange in the order of speed - HashMap, HashTable, Collections.synchronizedMap,concurrentHashmap?**

HashMap is fastest, ConcurrentHashMap,Collections.synchronizedMap,HashTable.

**What is IdentityHashMap?**

The IdentityHashMap uses == for equality checking instead of equals(). This can be used for both performance reasons, if you know that two different elements will never be equals and for preventing spoofing, where an object tries to imitate another.

**What is WeakHashMap?**

A hashtable-based Map implementation with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently than other Map implementations.

**How to call a store procedure in JAVA ?**

Connection dbConnection = null;

CallableStatement callableStatement = null;

String getDBUSERByUserIdSql = "{call getDBUSERByUserId(?,?,?,?)}";

try {

dbConnection = getDBConnection();

callableStatement = dbConnection.prepareCall(getDBUSERByUserIdSql);

callableStatement.setInt(1, 10);

callableStatement.registerOutParameter(2, java.sql.Types.VARCHAR);

callableStatement.registerOutParameter(3, java.sql.Types.VARCHAR);

callableStatement.registerOutParameter(4, java.sql.Types.DATE);

// execute getDBUSERByUserId store procedure

callableStatement.executeUpdate();

String userName = callableStatement.getString(2);

String createdBy = callableStatement.getString(3);

Date createdDate = callableStatement.getDate(4);

System.out.println("UserName : " + userName);

System.out.println("CreatedBy : " + createdBy);

System.out.println("CreatedDate : " + createdDate);

} catch (SQLException e) {

System.out.println(e.getMessage());

} finally {

if (callableStatement != null) {

callableStatement.close();

}

if (dbConnection != null) {

dbConnection.close();

}

}

}